Errata Sheet

Rel. 1.3, 03.04.2009

Device	**TC1130**
Marking/Step	**BB**
Package	**PG-LBGA-208**

This Errata Sheet describes the deviations from the current user documentation.

Table 1 Current Documentation

TC1130 User's Manual System Units	V1.3	Nov 2004
TC1130 User's Manual Peripheral Units	V1.3	Nov 2004
TC1130 Data Sheet	V1.1	Dec 2008
TriCore 1 Architecture	V1.3.8	Nov 2007

Each erratum identifier follows the pattern Module_Arch.TypeNumber:

- **Module**: subsystem or peripheral affected by the erratum
- **Arch**: microcontroller architecture where the erratum was firstly detected.
  - **AI**: Architecture Independent (detected on module level)
  - **CIC**: Companion ICs
  - **TC**: TriCore (32 bit)
  - **X**: XC1xx / XC2xx (16 bit)
  - **XC8**: XC8xx (8 bit)
  - **none**: C16x (16 bit)
- **Type**: none - Functional Deviation; '**P**' - Parametric Deviation; '**H**' - Application Hint; '**D**' - Documentation Update
- **Number**: ascending sequential number within the three previous fields. As this sequence is used over several derivatives, including already solved deviations, gaps inside this enumeration can occur.

Note: Devices marked with EES or ES are engineering samples which may not be completely tested in all functional and electrical characteristics, therefore they should be used for evaluation only.

Note: This device is equipped with a TriCore "TC1.3" Core. Some of the errata have a workaround which is possibly supported by the compiler tool vendor. Some corresponding compiler switches need possibly to be set. Please see the respective documentation of your compiler.

The specific test conditions for EES and ES are documented in a separate Status Sheet.

1 History List / Change Summary

Table 2 History List

Version	Date	Remark
1.0	22.07.2005	
1.1	19.12.2005	
1.2	01.02.2006	

Table 3 Errata fixed in this step

Errata	Short Description	Chg
DMI_TC.012	Data corruption possible during load from data cache	Fixed
DMI_TC.013	Data corruption possible when accessing data cache	Fixed
MLI_TC.003	MLI handles RETRY on FPI bus incorrectly	Fixed
MLI_TC.004	Read frame data may be corrupt when FPI error occured	Fixed
MultiCAN_TC.021	Wrong MultiCAN identifier-transmission	Fixed
MultiCAN_TC.022	MultiCAN error passive	Fixed
MultiCAN_TC.023	Disturbed transmit filtering	Fixed
PORT_TC.H003	Internal pull up is not working during reset	Fixed
SSC_TC.006	Leading delay for SLSOx stalls SSCx	Fixed
SSC_TC.007	Unintended switching of slave-selects in SSC0	Fixed
SCU_TC.008	Boot from CAN is not stable	Fixed
USB_TC.002	Not possible to send Zero Length Packets (ZLP)	Fixed

Table 3 Errata fixed in this step (cont'd)

Errata	Short Description	Chg
USB_TC.003	FIFO NOT READY interrupt sent by USB_11d during shadow register writeback of CPU and USB	Fixed

Table 4 Functional Deviations

Functional Deviation	Short Description	Chg	Pg
BCU_TC.002	SBCU does not give bus error		13
CPU_TC.004	CPU can be halted by writing DBGSR with OCDS Disabled		13
CPU_TC.008	IOPC Trap taken for all un-acknowledged Co-processor instructions		13
CPU_TC.012	Definition of PACK and UNPACK fail in certain corner cases		14
CPU_TC.013	Unreliable context load/store operation following an address register load instruction		15
CPU_TC.014	Wrong rounding in 8000*8000<<1 case for certain MAC instructions		16
CPU_TC.046	FPI master livelock when accessing reserved areas of CSFR space		16
CPU_TC.048	CPU fetches program from unexpected address		17
CPU_TC.052	Alignment Restrictions for Accesses using PTE-Based Translation		18
CPU_TC.053	PMI line buffer is not invalidated during CPU halt		19
CPU_TC.056	Incorrect probe.i operation in MMU UTLB		19

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
CPU_TC.059	Idle Mode Entry Restrictions		20
CPU_TC.060	LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results		21
CPU_TC.061	Error in emulator memory protection override		22
CPU_TC.062	Error in circular addressing mode for large buffer sizes		23
CPU_TC.063	Error in advanced overflow flag generation for SHAS instruction		24
CPU_TC.064	Co-incident FCU and CDO traps can cause system-lock		25
CPU_TC.065	Error when unconditional loop targets unconditional jump		26
CPU_TC.066	Incorrect forwarding when dependent CACHEA follows LD.[D]A		27
CPU_TC.067	Incorrect operation of STLCX instruction		27
CPU_TC.068	Potential PSW corruption by cancelled DVINIT instructions		28
CPU_TC.069	Potential incorrect operation of RSLCX instruction		30
CPU_TC.070	Error when conditional jump precedes loop instruction		31
CPU_TC.071	Error when Conditional Loop targets Unconditional Loop		32
CPU_TC.072	Error when Loop Counter modified prior to Loop instruction		32
CPU_TC.073	Debug Events on Data Accesses to Segment E/F Non-functional		33

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
CPU_TC.074	Interleaved LOOP/LOOPU instructions may cause GRWP Trap		33
CPU_TC.075	Interaction of CPS SFR and CSFR reads may cause livelock		35
CPU_TC.077	CACHEA.I instruction executable in User Mode		36
CPU_TC.078	Possible incorrect overflow flag for an MSUB.Q and an MADD.Q instruction		36
CPU_TC.079	Possible invalid ICR.PIPN when no interrupt pending		37
CPU_TC.080	No overflow detected by DVINIT instruction for MAX_NEG / -1		38
CPU_TC.081	Error during Load A[10], Call / Exception Sequence		39
CPU_TC.082	Data corruption possible when Memory Load follows Context Store		40
CPU_TC.083	Interrupt may be taken following DISABLE instruction		41
CPU_TC.085	CPS module may error acknowledge valid read transactions		42
CPU_TC.086	Incorrect Handling of PSW.CDE for CDU trap generation		43
CPU_TC.087	Exception Prioritisation Incorrect		44
CPU_TC.088	Imprecise Return Address for FCU Trap		46
CPU_TC.089	Interrupt Enable status lost when taking Breakpoint Trap		47
CPU_TC.090	MMU Page Table Entry Mapping Restrictions		48
CPU_TC.091	Incorrect privilege handling of MMU instructions		49

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
CPU_TC.092	Upper Memory Segments accessible in User-0 Mode with MMU enabled		50
CPU_TC.093	MMU Instruction Usage Restrictions		50
CPU_TC.094	Potential Performance Loss when CSA Instruction follows IP Jump		51
CPU_TC.095	Incorrect Forwarding in SAT, Mixed Register Instruction Sequence		52
CPU_TC.096	Error when Conditional Loop targets Single Issue Group Loop		54
CPU_TC.097	Overflow wrong for some Rounding Packed Multiply-Accumulate instructions.	New	55
CPU_TC.098	Possible PSW.V Error for an MSUB.Q instruction variant when both multiplier inputs are of the form 0x8000xxxx	New	56
CPU_TC.099	Saturated Result and PSW.V can error for some q format multiply-accumulate instructions when computing multiplications of the type 0x80000000*0x8000 when n=1	New	57
CPU_TC.100	Mac instructions can saturate the wrong way and have problems computing PSW.V	New	65
CPU_TC.101	MSUBS.U can fail to saturate result, and MSUB(S).U can fail to assert PSW.V	New	70
CPU_TC.102	Result and PSW.V can be wrong for some rounding, packed, saturating, MAC instructions.	New	72
CPU_TC.103	Spurious parity errors can be generated	New	73
CPU_TC.104	Double-word Load instructions using Circular Addressing mode can produce unreliable results	New	74

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
CPU_TC.105	User / Supervisor mode not staged correctly for Store Instructions	New	76
CPU_TC.107	SYSCON.FCDSF may not be set after FCD Trap	New	77
CPU_TC.108	Incorrect Data Size for Circular Addressing mode instructions with wrap-around	New	78
CPU_TC.109	Circular Addressing Load can overtake conflicting Store in Store Buffer	New	81
CPU_TC.112	Unreliable result for MFCR read of Program Counter (PC)	New	84
DMA_TC.004	Reset of registers OCDSR and SUSPMR is connected to FPI reset		85
DMA_TC.005	Do not access MExPR, MExAENR, MExARR with RMW instructions		86
DMA_TC.007	CHSRmn.LXO bit is not reset by channel reset		86
DMA_TC.010	Channel reset disturbed by pattern found event		86
DMA_TC.011	Pattern search for unaligned data fails on certain patterns		87
DMA_TC.012	No wrap around interrupt generated		87
DMI_TC.005	DSE Trap possible with no corresponding flag set in DMI_STR		88
Ethernet_TC.007	Incorrect MAC behavior when handling dribble bits		88
Ethernet_TC.008	IData might be lost when descriptor contains data less than 14 bytes in multiple-descriptor frame		89

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
FPU_TC.001	FPU flags always update with FPU exception		89
IIC_AI.001	Handling of the Receive/Transmit Buffer RTB	New	89
IIC_AI.003	SCL low time tLOW violated before repeated start condition in standard mode	New	90
IIC_AI.005	Restart condition only possible with CI = 0	New	90
MLI_TC.006	Receiver address is not wrapped around in downward direction		91
MLI_TC.007	Answer frames do not trigger NFR interrupt if RIER.NFRIE=10_B and Move Engine enabled		92
MLI_TC.008	Move engines can not access address $F01E0000_H$		92
MLI_TC.009	MLI0B and internal loopback option not available for TC1130.		92
MultiCAN_AI.040	Remote frame transmit acceptance filtering error	New	93
MultiCAN_AI.041	Dealloc Last Obj	New	94
MultiCAN_AI.042	Clear MSGVAL during transmit acceptance filtering	New	94
MultiCAN_AI.043	Dealloc Previous Obj	New	94
MultiCAN_AI.044	RxFIFO Base SDT	New	95
MultiCAN_AI.045	OVIE Unexpected Interrupt	New	96
MultiCAN_AI.046	Transmit FIFO base Object position	New	96
MultiCAN_TC.024	Power-on recovery		96
MultiCAN_TC.025	RXUPD behavior		100
MultiCAN_TC.026	MultiCAN Timestamp Function		100
MultiCAN_TC.027	MultiCAN Tx Filter Data Remote		101

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
MultiCAN_TC.028	SDT behavior		101
MultiCAN_TC.029	Tx FIFO overflow interrupt not generated		102
MultiCAN_TC.030	Wrong transmit order when CAN error at start of CRC transmission		104
MultiCAN_TC.031	List Object Error wrongly triggered	New	104
MultiCAN_TC.032	MSGVAL wrongly cleared in SDT mode	New	105
MultiCAN_TC.035	Different bit timing modes		105
MultiCAN_TC.037	Clear MSGVAL	New	107
MultiCAN_TC.038	Cancel TXRQ	New	108
OCDS_TC.007	DBGSR writes fail when coincident with a debug event		108
OCDS_TC.008	Breakpoint interrupt posting fails for ICR modifying instructions		110
OCDS_TC.009	Data access trigger events unreliable		110
OCDS_TC.010	DBGSR.HALT[0] fails for separate resets		110
OCDS_TC.011	Context lost for multiple breakpoint traps		111
OCDS_TC.012	Multiple debug events on one instruction can be unpredictable		112
PMI_TC.001	Deadlock possible during Instruction Cache Invalidation		112
SSC_AI.023	Clock phase control causes failing data transmission in slave mode	New	113
SSC_AI.024	SLSO output gets stuck if a reconfig from slave to master mode happens	New	113
SSC_AI.025	First shift clock period will be one PLL clock too short because not syncronized to baudrate	New	113
SSC_AI.026	Master with highest baud rate set generates erroneous phase error	New	114

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Chg	Pg
SSC_TC.008	SSC shift register not updated in fractional divider mode		114
SSC_TC.011	Unexpected phase error		115
SSC_TC.017	Slaveselect (SLSO) delays may be ignored	New	116

Table 5 Deviations from Electrical- and Timing Specification

AC/DC/ADC Deviation	Short Description	Chg	Pg

Table 6 Application Hints

Hint	Short Description	Chg	Pg
IIC_AI.H003	IIC master cannot lose arbitration at acknowledge bit during read	New	118
IIC_AI.H005	General call feature does not work with 10-bit address mode	New	118
IIC_AI.H006	TRX- Bit is not immediately set after writing to register RTB	New	118
INT_TC.H001	Multiple SRNs can be assigned to the same SRPN (priority)		118
MultiCAN_AI.H005	TxD Pulse upon short disable request	New	119
MultiCAN_TC.H002	Double Synchronization of receive input	New	119
MultiCAN_TC.H003	Message may be discarded before transmission in STT mode	New	119
MultiCAN_TC.H004	Double remote request	New	120

Table 6 Application Hints (cont'd)

Hint	Short Description	Chg	Pg
SSC_AI.H002	Transmit Buffer Update in Master Mode during Trailing or Inactive Delay Phase	New	121
SSC_AI.H003	Transmit Buffer Update in Slave Mode during Transmission	New	121
SSC_TC.H002	Enlarged leading delay in master mode		122

2 Functional Deviations

BCU_TC.002 SBCU does not give bus error

SBCU does not give bus error when the following memory segment is accessed:

0-7
9,B & C

and memory address range of:

0xD0000000 - 0xD0007FFF

Workaround

None.

CPU_TC.004 CPU can be halted by writing DBGSR with OCDS Disabled

Contrary to the specification, the TriCore1 CPU can be halted by writing "11" to the DBGSR.HALT bits, irrespective of whether On-Chip Debug Support (OCDS) is enabled or not (DBGSR.DE not checked).

Workaround

None.

CPU_TC.008 IOPC Trap taken for all un-acknowledged Co-processor instructions

When the TriCore1 CPU encounters a co-processor instruction, the instruction is routed to the co-processor interface where further decoding of the opcode is performed in the attached co-processors. If no co-processor acknowledges that this is a valid instruction, the CPU generates an illegal opcode (IOPC) trap.

Early revisions of the TriCore Architecture Manual are unclear regarding whether Un-Implemented OPCode (UOPC) or Invalid OPCode (IOPC) traps should be taken for un-acknowledged co-processor instructions. However, the required behaviour is that instructions routed to a given co-processor, where the co-processor is present but does not understand the instruction opcode, should result in an IOPC trap. Co-processor instructions routed to a co-processor, where that co-processor is not present in the system, should result in a UOPC trap.

Consequently the current TriCore1 implementation does not match the required behaviour in the case of un-implemented co-processors.

Workaround

Where software emulation of un-implemented co-processors is required, the IOPC trap handler must be written to perform the required functionality.

CPU_TC.012 Definition of PACK and UNPACK fail in certain corner cases

Revisions of the TriCore Architecture Manual, up to and including V1.3.3, do not consistently describe the behaviour of the PACK and UNPACK instructions. Specifically, the instruction definitions state that no special provision is made for handling IEEE-754 denormal numbers, infinities, NaNs or Overflow/Underflow situations for the PACK instruction. In fact, all these special cases are handled and will be documented correctly in further revisions of the TriCore Architecture Manual.

However, there are two situations where the current TriCore1 implementation is non-compliant with the updated definition, as follows:

1. Definition and detection of Infinity/NaN for PACK and UNPACK

In order to avoid Infinity/NaN encodings overlapping with arithmetic overflow situations, the special encoding of un-biased integer exponent = 255 and high order bit of the normalized mantissa (bit 30 for UNPACK, bit 31 for PACK) = 0 is defined.

In the case of Infinity or NaN, the TriCore1 implementation of UNPACK sets the un-biased integer exponent to +255, but sets the high order bit of the

normalized mantissa (bit 30) to 1. In the case of PACK, input numbers with biased exponent of 255 and the high order bit of the normalized mantissa (bit 31) set are converted to Infinity/NaN. Unfortunately, small overflows may therefore be incorrectly detected as NaN by the PACK instruction special case logic and converted accordingly, when an overflow to Infinity should be detected.

2. Special Case Detection for PACK

In order to detect special cases, the exponent is checked for certain values. In the current TriCore1 implementation this is performed on the biased exponent, i.e. after 128 has been added to the un-biased exponent. In the case of very large overflows the addition of 128 to the un-biased exponent can cause the exponent itself to overflow and be interpreted as a negative number, i.e. underflow, causing the wrong result to be produced.

Workaround

The corner cases where the PACK instruction currently fails may be detected and the input number re-coded accordingly to produce the desired result.

CPU_TC.013 Unreliable context load/store operation following an address register load instruction

When an address register is being loaded by a load/store instruction LD.A/LD.DA and this address register is being used as address pointer in a following context load/store instruction LD*CX/ST*CX it may lead to unpredictable behavior.

Example

```
...
LD.A   A3, <any addressing mode>; load value into A3
LDLCX  [A3]   ; context load
...
```

Workaround

Insert one NOP instruction between the address register load/store instruction and the context load/store instruction to allow the "Load Word to Address Register" operation to be completed first.

```
...
LD.A  A3, <any addressing mode>
NOP
LDLCX [A3]
...
```

CPU_TC.014 Wrong rounding in 8000*8000<<1 case for certain MAC instructions

In the case of "round(acc +/- 8000_H * 8000_H << 1)" the multiplication and the following accumulation is carried out correctly. However, rounding is incorrect.

Rounding is done in two steps:

1. Adding of 0000 8000_H
2. Truncation

For the before mentioned case the first step during rounding (i.e. the adding operation) is suppressed - which is wrong - while truncation is carried out correctly.

This bug affects all variants of MADDR.Q, MADDR.H, MSUBR.Q, MSUBR.H., MADDSUR.H and MSUBADR.H instructions.

Workaround

None.

CPU_TC.046 FPI master livelock when accessing reserved areas of CSFR space

The Core Special Function Registers (CSFRs) associated with the TriCore1 CPU are accessible by any FPI bus master, other than the CPU, in the address range F7E1 0000_H - F7E1 $FFFF_H$. Any access to an address within this range

which does not correspond to an existing CSFR within the CPU may result in the livelock of the initiating FPI master.

Accesses to the CPU CSFR space are performed via the CPU's slave interface (CPS) module, by any FPI bus master other than the CPU itself. In the case of such an access the CPS module initially issues a retry acknowledge to the FPI master then injects an instruction into the CPU pipeline to perform the CSFR access. The initial access is retry acknowledged to ensure the FPI bus is not blocked and instructions in the CPU pipeline are able to progress. The CPS module will continue to retry acknowledge further attempts by the FPI master to read the CSFR until the data is returned by the CPU.

In the case of an access to a reserved CSFR location the CPU treats the instruction injected by the CPS as a NOP and never acknowledges the CSFR access request. As such the CPS module continues to retry the CSFR access on the FPI bus, leading to the lockup of the initiating FPI master.

Workaround

Do not access reserved areas of the CPU CSFR space.

CPU_TC.048 CPU fetches program from unexpected address

There is a case which can cause the CPU to fetch program code from an unexpected address. Although this code will not be executed the program fetch itself can cause side effects (performance degradation, program fetch bus error trap).

If a load address register instruction LD.A/LD.DA is being followed immediately by an indirect jump JI, JLI or indirect call CALLI instruction with the same address register as parameter, the CPU might fetch program from an unexpected address.

Workaround

Insert a NOP instruction or any other load/store instruction between the load and the indirect jump/call instruction. (See also note `Pipeline Effects`, below)

Example

```
...
LD.A          A14, <any addressing mode>
NOP           ; workaround to prevent program
              ; fetch from undefined address
<one optional IP instruction>
CALLI         A14
...
```

Pipeline Effects

The CPU core architecture allows to decode and execute instructions for the integer pipeline (IP) and the load/store pipeline (LS) in parallel. Therefore this bug hits also if there is only one IP instruction sitting in front of the offending LS instruction (`CALLI A14` in above example). A detailed list of IP instructions can be found in the document `TriCore DSP Optimization Guide - Part 1: Instruction Set, Chapter 13.1.3, Table of Dual Issue Instructions".

CPU_TC.052 Alignment Restrictions for Accesses using PTE-Based Translation

Additional alignment restrictions exist for TriCore load-store accesses which undergo PTE-based translation.For devices which include the optional Memory Management Unit (MMU), accesses to a virtual address in one of the lower 8 segments of the address space, where the processor is operating in virtual mode (MMU enabled), undergo PTE-based translation.

For such accesses, the cacheability of the resultant memory access depends upon both the cacheability attribute of the resultant physical address and the cacheability flag of the PTE used to perform the translation. Only when the resultant physical address is cacheable and the PTE cacheability flag is set will the access be cacheable.

For load-store accesses undergoing PTE-based translation the assumption is made that the resultant access is to a cacheable location and that no special handling of the mis-aligned access is required. If the resultant access, after PTE

transaltion, is non-cacheable and not naturally aligned, then a Data Address Alignment (ALN) trap will be generated.

Workaround

Natural alignment must be used for accesses undergoing PTE-based translation which may result in a non-cacheable memory access.

CPU_TC.053 PMI line buffer is not invalidated during CPU halt

Some debug tools provide the feature to modify the code during runtime in order to realize breakpoints. They exchange the instruction at the breakpoint address by a 'debug' instruction, so that the CPU goes into halt mode before it passes the instruction. Thereafter the debugger replaces the debug instruction by the original code again.

This feature no longer works reliably as the line buffer will not be invalidated during a CPU halt. Instead of the original instruction, the obsolete debug instruction will be executed again.

Workaround

Debuggers might use the following macro sequence:

1. set PC to other memory address (> 0x20h, which selects new cacheline-refill buffer)
2. execute at least one instruction (e.g. NOP) and stop execution again (e.g. via debug instruction)
3. set PC back to former debug position
4. proceed execution of user code

CPU_TC.056 Incorrect probe.i operation in MMU UTLB

The TLBPROBE.I instruction takes a data register, `D[a]`, as a parameter and uses it to probe the MMU Translation Lookaside Buffer (TLB) at a given index. The `D[a]` register contains the index for the probe. The results of the TLBPROBE.I instruction are placed in the TVA and TPA Core Special Function

Registers (CSFRs). Under certain circumstances the TLBPROBE.I instruction may fail and return the result from an incorrect index.

The problem occurs if the unused fields of D[a] match a VPN for a different index in the TLB. In this case the TLB hit logic is incorrectly activated and the attributes from the index with the matching VPN read.

Workaround

The unused fields of D[a] should be set to '1' to avoid any erroneous VPN matches in the UTLB. For example, if the index required to be probed is 0x80, the actual value 0x00000080 should not be placed in D[a], rather 0xFFFFFF80 should be used.

CPU_TC.059 Idle Mode Entry Restrictions

Two related problems exist which lead to unreliable idle mode entry, and possible data corruption, if the idle request is received whilst the TriCore CPU is in certain states. The two problems are as follows:

1. When the TriCore CPU receives an idle request, a DSYNC instruction is injected to flush any data currently held within the CPU to memory. If there is any outstanding context information to be saved, the clocks may be disabled too early, before the end of the context save. The CPU is then frozen in an erroneous state where it is instructing the DMI to make continuous write accesses onto the bus. Because of the pipelined architecture, the DMI may also see the wrong address for the spurious write accesses, and therefore memory data corruption can emerge. Another consequence of this is, that the DMI will not go to sleep and therefore the IDLE-state will not be fully entered.
2. If the idle request is asserted when a DSYNC instruction is already being executed by the TriCore CPU, the idle request may be masked prematurely and the idle request never acknowledged.

Workaround

```
...
DISABLE                    ; Disable Interrupts NOP
```

```
DSYNC                    ; Flush Buffers, background context
ISYNC                      ; Ensure DSYNC completes
<Store to SCU to assert idle request>
NOP                        ; Wait on idle request
NOP                        ; Wait on idle request
...
```

CPU_TC.060 LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results

An LD.A or LD.DA instruction followed back to back by an unaligned LD.DA, LD.D or LD.W instruction can lead to unreliable results. This problem is independent of the instruction formats (16 and 32 bit versions of both instructions are similarly affected)

The problem shows up if the LD.DA, LD.D or LD.W uses an address register which is loaded by the preceding LD.A or LD.DA and if the LD.DA, LD.D or LD.W accesses data which leads to a multicycle execution of this second instruction.

A multicycle execution of LD.DA, LD.D or LD.W will be triggered only if the accessed data spans a 128 bit boundary in the local DSPR space or a 128 bit boundary in the cached space. In the non cached space an access spanning a 64 bit boundary can lead to a multicycle execution.

The malfunction is additionally dependent on the previous content of the used address register - the bug appears if the content points to the unimplemented DSPR space.

In the buggy case the upper portion of the multicycle load is derived from a wrong address (the address is dependent on the previous content of that address register) and the buggy case leads to a one cycle FASTER execution of this back to back case. (one stall bubble is lacking in this case)

The 16 and 32 bit variants of both instructions are affected equally. A single IP instruction as workaround is NOT sufficient, as it gets dual issued with the LD.[DA,D,W] and therefore no bubble is seen by the LS pipeline in such a case.

Example:

```
...
```

```
LD.A    A3,<any addressing mode>; load pointer into A3
LD.W    D1,[A3]<any addressing mode>; load data value from
                                     ; pointer
...
```

Workaround

Insert one NOP instruction between the address register load/store instruction and the data load/store instruction to allow the "Load Word to Address Register" operation to be completed first. This leads to a slight performance degradation.

```
...
LD.A    A3, <any addressing mode>
NOP
LD.W    D1, [A3] <any addressing mode>
...
```

Alternative Workaround

To avoid the slight performance degradation, an alternative workaround is to avoid any data structures that are accessed in an unaligned manner as then the described instruction sequence does NOT exhibit any problems.

CPU_TC.061 Error in emulator memory protection override

TriCore1 based systems define an area of the system address map for use as an emulator memory region. Whenever a breakpoint trap is taken, the processor jumps to the base of this emulator region from where a debug monitor is executed.

In order to allow correct execution of this monitor, in the presence of an enabled protection system, this emulator region is granted implicit execute permission. Execution of code from this region is allowed whether the current settings of the memory protection ranges specifically permit this or not, and no MPX trap will be generated.

In TriCore1.2 based systems, this emulator memory region existed at addresses 0xBExxxxxx. In TriCore1.3 based systems, this emulator region

initially was moved to addresses 0xDExxxxxx before being made fully programmable.

The erroneous behaviour occurs because as this emulator region was moved from addresses 0xBExxxxxx, the implicit execute permission to this address range was not moved also. As a result:

1. Code execution from addresses in the range 0xBE000000 - 0xBEFFFFFF is always permitted, irrespective of the settings of the protection system.
2. Execution of a breakpoint trap may result in the generation of an MPX trap if execution from the new emulator region is dis-allowed by the current settings of the protection system.

Workaround

None

CPU_TC.062 Error in circular addressing mode for large buffer sizes

A problem exists in the circular addressing mode when large buffer sizes are used. Specifically, the problem exists when:

1. The length, L, of the circular buffer is >=32768 bytes, i.e. MSB of L is '1'

AND

2. The offset used to access the circular buffer is negative.

In this case the update of the circular buffer index may be calculated incorrectly and the addressing mode fail.

Each time an instruction using circular addressing mode occurs the next index for the circular buffer is calculated as current index + offset, where the signed offset is supplied as part of the instruction. In addition, the situation where the new index lies outside the bounds of the circular buffer has to be taken care of and the correct wrapping behaviour performed. In the case of negative offsets, the buffer underflow condition needs to be checked and, when detected, the buffer size is added to the index in order to implement the required wrapping.

Due to an error in the way the underflow condition is detected, there are cases where the buffer size is incorrectly added to the index when there is no buffer

underflow. This false condition is detected when the index is greater than or equal to 32768 and the offset is negative.

Example:

```
...
MOVH.A A1, #0xE001          ;
LEA    A1, [A1]-0x4000      ; Buffer Length 0xE000,
                            ; Index 0xC000
LEA    A0, 0xA0000000       ; Buffer Base Address
LD.W   D9, [A0/A1+c]-0x4    ; Circular addressing
                            ; mode access,
                            ; negative offset
...
```

Workaround

Either limit the maximum buffer size for circular addressing mode to 32768 bytes, or use only positive offsets where larger circular buffers are required.

CPU_TC.063 Error in advanced overflow flag generation for SHAS instruction

A minor problem exists with the computation of the advanced overflow (AV) flag for the SHAS (Arithmetic Shift with Saturation) instruction. The TriCore architecture defines that for instructions supporting saturation, the advanced overflow flag shall be computed BEFORE saturation. The implementation of the SHAS instruction is incorrect with the AV flag computed after saturation.

Example:

```
...
MOVH   D0, #0x4800          ; D0 = 0x48000000
MOV.U  D1, #0x2             ; D1 = 0x2
SHAS   D2, D0, D1           ; Arithmetic Shift
                            ; with Saturation
...
```

In the above example, the result of 0x4800_0000 << 2 = 0x1_2000_0000, such that the expected value for AV = bit31 XOR bit30 = 0. However, after saturation the result is 0x7FFF_FFFF and the AV flag is incorrectly set.

Workaround

None

CPU_TC.064 Co-incident FCU and CDO traps can cause system-lock

A problem exists in the interaction between Free Context Underflow (FCU) and Call Depth Overflow (CDO) traps. An FCU trap occurs when a context save operation is attempted and the free context list is empty, or when the context operation encounters an error. A CDO trap occurs when a program attempts to make a call with call depth counting enabled and the call depth counter was already at its maximum value.

When an FCU trap occurs with call depth counting enabled (PSW.CDE = '1') and the call depth counter at a value such that the next call will generate a CDO trap, then the FCU trap causes a co-incident CDO trap. In this case the PC is correctly set to the FCU trap handler but appears to freeze in this state as a constant stream of FCU traps is generated.

A related problem occurs when call trace mode is enabled (PSW.CDC = 0x7E). If in call trace mode a call or return operation encounters an FCU trap, either a CDO (call) or Call Depth Underflow (CDU, return) trap is generated co-incident with the FCU trap, either of which situations lead to a constant stream of FCU traps and system lockup.

Note however that FCU traps are not expected during normal operation since this trap is indicative of software errors.

Workaround

None

CPU_TC.065 Error when unconditional loop targets unconditional jump

An error in the program flow occurs when an unconditional loop (LOOPU) instruction has as its target an unconditional jump instruction, i.e. as the first instruction of the loop. Such unconditional jump instructions are J, JA, JI, JL, JLA and JLI.

In this erroneous case the first iteration of the loop executes correctly. However, at the point the second loop instruction is executed the interaction of the unconditional loop and jump instructions causes the loop instruction to be resolved as mis-predicted and the program flow exits the loop incorrectly, despite the loop instruction being unconditional.

Example:

```
...
loop_start_:          ; Loop start label
J jump_label_         ; Unconditional Jump instruction
...
LOOPU loop_start_
...
```

Workaround

The first instruction of a loop may not be an unconditional jump. If necessary, precede this jump instruction with a single NOP.

```
...
loop_start_:          ; Loop start label
NOP
J jump_label_         ; Unconditional Jump instruction
...
LOOPU loop_start_
...
```

CPU_TC.066 Incorrect forwarding when dependent CACHEA follows LD.[D]A

An error can occur when an LD.A or LD.DA instruction is followed back to back by a data cache management instruction (CACHEA.W, CACHEA.WI or CACHEA.I). The problem occurs if the addressing mode of the cache management instruction uses the address register which is being loaded by the preceding LD.A or LD.DA instruction. A problem exists in the logic required to detect the read after write hazard between these two instructions, which may lead to the old value of the address register being used erroneously for the CACHEA instruction.

Example:

```
...
LD.A   A3, <any addressing mode>
CACHEA.W[A3] <any addressing mode>
...
```

Workaround

Insert one NOP instruction between the address register load instruction and the data cache management instruction to allow the "Load Word to Address Register" operation to be completed first.

```
...
LD.A   A3, <any addressing mode>
NOP
CACHEA.W[A3] <any addressing mode>
...
```

CPU_TC.067 Incorrect operation of STLCX instruction

There is an error in the operation of the Store Lower Context (STLCX) instruction. This instruction stores the current lower context information to a 16-word memory block specified by the addressing mode associated with the instruction (not to the free context list). The architectural definition of the STLCX instruction is as follows:

Mem(EA, 16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]}

However, there is an error in the implementation of the instruction, such that the following operation is actually performed:

Mem(EA, 16-word) = {PCXI, **PSW**, A[2:3], D[0:3], A[4:7], D[4:7]}

i.e. the PSW is incorrectly stored instead of A11.

During normal operation, the lower context information that has been stored by an STLCX instruction would be re-loaded using the Load Lower Context (LDLCX) operation. The architectural definition of the LDLCX instruction is as follows:

{-, -, A[2:3], D[0:3], A[4:7], D[4:7]} = Mem(EA, 16-word)

i.e. the value which is incorrectly stored by STLCX is not re-loaded by LDLCX, such that the erroneous behaviour is not seen during normal operation.

However, any attempt to reload a lower context stored with STLCX using load instructions other than LDLCX will exhibit the incorrect behaviour.

Workaround

Any lower context stored using STLCX should only be re-loaded using LDLCX, otherwise the erroneous behaviour will be visible.

CPU_TC.068 Potential PSW corruption by cancelled DVINIT instructions

A problem exists in the implementation of the Divide Initialisation instructions, which, under certain circumstances, may lead to corruption of the advanced overflow (AV), overflow (V) and sticky overflow (SV) flags. These flags are stored in the Program Status Word (PSW) register, fields PSW.AV, PSW.V and PSW.SV. The divide initialisation instructions are DVINIT, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.H and DVINIT.HU.

The problem is that the DVINIT class instructions do not handle the instruction cancellation signal correctly, such that cancelled DVINIT instructions still update the PSW fields. The PSW fields are updated according to the operands supplied to the cancelled DVINIT instruction. Due to the nature of the DVINIT instructions this can lead to:

- The AV flag may be negated erroneously.

- The V flag may be asserted or negated erroneously.
- The SV flag may be asserted erroneously.

No other fields of the PSW can be affected. A DVINIT class instruction could be cancelled due to a number of reasons:

- the DVINIT instruction is cancelled due to a mis-predicted branch
- the DVINIT instruction is cancelled due to an unresolved operand dependency
- the DVINIT instruction is cancelled due to an asynchronous event such as an interrupt

Workaround

If the executing program is using the PSW fields to detect overflow conditions, the correct behaviour of the DVINIT instructions may be guaranteed by avoiding the circumstances which could lead to a DVINIT instruction being cancelled. This requires that the DVINIT instruction is preceded by 2 NOPs (to avoid operand dependencies or the possibility of mis-predicted execution). In addition, the status of the interrupt enable bit ICR.IE must be stored and interrupts disabled before the 2 NOPs and the DVINIT instruction are executed, and the status of the ICR.IE bit restored after the DVINIT instruction is complete.

Alternative Workaround

To avoid the requirement to disable and re-enable interrupts an alternative workaround is to precede the DVINIT instruction with 2 NOPs and to store the PSW.SV flag before a DVINIT instruction and check its consistency after the DVINIT instruction. In this case the values of the PSW flags affected may be incorrect whilst the asynchronous event is handled, but once the return from exception is complete and the DVINIT instruction re-executed, only the SV flag can be in error. In this case if the SV flag was previously negated but after the DVINIT instruction the SV flag is asserted and the V flag is negated, then the SV flag has been asserted erroneously and should be corrected by software.

CPU_TC.069 Potential incorrect operation of RSLCX instruction

A problem exists in the implementation of the RSLCX instruction, which, under certain circumstances, may lead to data corruption in the TriCore internal registers. The problem is caused by the RSLCX instruction incorrectly detecting a dependency to the following load-store (LS) or loop (LP) pipeline instruction, if that instruction uses either address register A0 or A1 as a source operand, and erroneous forwarding paths being enabled.

Two failure cases are possible:

1. If the instruction following the RSLCX instruction uses A1 as its source 1 operand, the PCX value updated by the RSLCX instruction will be corrupted. Instead of restoring the PCX value from the lower context information being restored, it will restore the return address (A11).
2. If the instruction following the RSLCX instruction uses either A1 or A0 as source 2 operand, the value forwarded (for the second instruction) will not be the one stored in the register but the one that has just been loaded from memory for the context restore (A11/PCX).

Note that the problem is triggered whenever the following load-store pipeline instruction uses A0 or A1 as a source operand. If an integer pipeline instruction is executed between the RSLCX and the following load-store or loop instruction, the problem may still exist.

Example:

```
...
RSLCX
LEA     A0, [A0]0x158c
...
```

Workaround

Any RSLCX instruction should be followed by a NOP to avoid the detection of these false dependencies.

CPU_TC.070 Error when conditional jump precedes loop instruction

An error in the program flow may occur when a conditional jump instruction is directly followed by a loop instruction (either conditional or unconditional). Both integer pipeline and load-store pipeline conditional jumps (i.e. those checking the values of data and address registers respectively) may cause the erroneous behaviour.

The incorrect behaviour occurs when the two instructions are not dual-issued, such that the conditional jump is in the execute stage of the pipeline and the loop instruction is at the decode stage. In this case, both the conditional jump instruction and the loop instruction will be resolved in the same cycle. The problem occurs because priority is given to the loop mis-prediction logic, despite the conditional jump instruction being semantically before the loop instruction in the program flow. In this error case the program flow continues as if the loop has exited: the PC is taken from the loop mis-prediction branch. In order for the erroneous behaviour to occur, the conditional jump must be incorrectly predicted as not taken. Since all conditional jump instructions, with the exception of 32-bit format forward jumps, are predicted as taken, only 32-bit forward jumps can cause the problem behaviour.

Example:

```
...
JNE.A  A1, A0, jump_target_1_ ; 32-bit forward jump
LOOP   A6, loop_target_1_
...
jump_target_1_:
...
```

Workaround

A conditional jump instruction may not be directly followed by a loop instruction (conditional or not). A NOP must be inserted between any load-store pipeline conditional jump (where the condition is dependent on an address register) and a loop instruction. Two NOPs must be inserted between any integer pipeline conditional jump (where the condition is dependent on a data register) and a loop instruction

CPU_TC.071 Error when Conditional Loop targets Unconditional Loop

An error in the program flow may occur when a conditional loop instruction (LOOP) has as its target an unconditional loop instruction (LOOPU). The incorrect behaviour occurs in certain circumstances when the two instructions are resolved in the same cycle. If the conditional loop instruction is mis-predicted, i.e. the conditional loop should be exited, the unconditional loop instruction is correctly cancelled but instead of program execution continuing at the first instruction after the conditional loop, the program flow is corrupted.

Example:

```
...
cond_loop_target_:
LOOPU  uncond_loop_target_   ; Unconditional loop
...
LOOP   A6, cond_loop_target_;Conditional loop targets
                            ;unconditional loop
...
```

Workaround

The first instruction of a conditional loop may not be an unconditional loop. If necessary, precede this unconditional loop instruction with a single NOP.

CPU_TC.072 Error when Loop Counter modified prior to Loop instruction

An error in the program flow may occur when an instruction that updates an address register is directly followed by a conditional loop instruction which uses that address register as its loop counter. The problem occurs when the address register holding the loop counter is initially zero, such that the loop will exit, but is written to a non-zero value by the instruction preceding the conditional loop. In this case the loop prediction logic fails and the program flow is corrupted.

Example:

```
...
LD.A   A6, <any addressing mode>
LOOP   A6, loop_target_1_
```

. . .

Workaround

Insert one NOP instruction between the instruction updating the address register and the conditional loop instruction dependent on this address register.

CPU_TC.073 Debug Events on Data Accesses to Segment E/F Non-functional

The generation of debug events from data accesses to addresses in Segments 0xE and 0xF is non-functional. As such the setting of breakpoints on data accesses to these addresses does not operate correctly.

In TriCore1 the memory protection system, consisting of the memory protection register sets and associated address comparators, is used both for memory protection and debug event generation for program and data accesses to specific addresses. For memory protection purposes, data accesses to the internal and external peripheral segments 0xE and 0xF bypass the range protection system and are protected instead by the I/O privilege level and protection mechanisms built in to the individual peripherals. Unfortunately this bypass of the range protection system for segments 0xE and 0xF also affects debug event generation, masking debug events for data accesses to these segments.

Workaround

None.

CPU_TC.074 Interleaved LOOP/LOOPU instructions may cause GRWP Trap

If a conditional loop instruction (LOOP) is executed after an unconditional loop instruction (LOOPU) a Global Register Write Protection (GRWP) Trap may be generated, even if the LOOP instruction does not use a global address register as its loop counter.

In order to support zero-overhead loop execution the TriCore1 implementation caches certain attributes pertaining to loop instructions within the CPU. The TriCore1.3 CPU contains two loop cache buffers such that two loop (LOOP or LOOPU) instructions may be cached. One of the attributes cached is whether the loop instruction writes to a global address register (as its loop variable). For LOOP instructions this attribute is updated and read as expected. For LOOPU instructions this attribute is set but ignored by the LOOPU instruction when next encountered.

The problem occurs because there is only one global address register write flag shared between the two loop caches. As such if LOOP and LOOPU instructions are interleaved, with the LOOPU instruction encountered and cached after the LOOP instruction, then the next execution of the LOOP instruction will find the global address register write flag set and, if global register writes are disabled (PSW.GW = 0), a GRWP trap will be incorrectly generated.

Example:

```
...
loopu_target_
...
loop_target_
...
LOOP   A5, loop_target_
...
LOOPU  loopu_target_
...
```

Workaround

Enable global register write permission, PSW.GW = 1.

Tool Vendor Workaround

The LOOPU instruction sets the global address register write flag when its un-used opcode bits [15:12] are incorrectly decoded as global address register A0. The problem may be avoided by assembling these un-used bits to correspond to a non-global register encoding, such as 0xF.

CPU_TC.075 Interaction of CPS SFR and CSFR reads may cause livelock

Under certain specific circumstances system lockup may occur if the TriCore CPU attempts to access a Special Function Register (SFR) within the CPS module around the same time as another master attempts to read a Core Special Function Register (CSFR), also via the CPS module.

In order to read a CSFR the CPS module injects an instruction into the CPU pipeline to access the required register. In order for this injected instruction to complete successfully the CPU pipeline must be allowed to progress. To avoid system lockup the CSFR read access is initially retry acknowledged on the FPI bus to ensure the FPI bus is not blocked and any CPU read access to an address mapped to the FPI bus is able to progress. The CPS then continues the CSFR read in the background, and, once complete, returns the data to the originating master when the read access is performed again.

The problem occurs if the CPU is attempting to access an SFR accessed via the CPS module around the time another master is attempting a CSFR read access. Under normal circumstances this causes no problem since the SFR access is allowed to complete normally even with an outstanding CSFR access in the background. However, if the SFR access is pipelined on the FPI bus behind the CSFR access and the CSFR access is still in progress then the interaction of the two pipelined transactions may cause the SFR access to be retry acknowledged in error. Thus the CPU pipeline is still frozen and the CSFR access cannot complete. As long as the two transactions, when re-initiated by their respective masters, continue to be pipelined on the FPI bus then this livelock situation will continue.

Note however that the only FPI master expected to access the CSFR address range via the CPS would be the Cerberus module under control of an external debugger. As such this livelock situation should only be possible whilst debugging, not during normal system operation.

Workaround

None.

CPU_TC.077 CACHEA.I instruction executable in User Mode

The CACHEA.W and CACHEA.WI instructions which writeback and optionally invalidate enties from the data cache are user mode executable instructions. The CACHEA.I instruction which invalidates data cache entries without writeback should be executable in supervisor mode only. However the current implementation is such that the CACHEA.I instruction is executable in user mode also.

Workaround

None.

CPU_TC.078 Possible incorrect overflow flag for an MSUB.Q and an MADD.Q instruction

Under certain conditions, a variant of the MSUB.Q instruction and a variant of the MADD.Q instruction can fail and produce an incorrect overflow flag, PSW.V, and hence an incorrect PSW.SV. When the problem behaviour occurs, the overflow flag is always generated incorrectly: if PSW.V should be set it will be cleared, and if it should be cleared it will be set.

The problem affects the following two instructions:

MSUB.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=63_H

MADD.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=43_H

The error conditions are as follows:

If (Da[31:16] = 16’h8000) and (DbL = 16’h8000) and (n=1), then PSW.V will be incorrect.

Workaround #1

If the PSW.V and PSW.SV flags generated by these instructions are not used by the code, then the instructions can be used without a workaround.

Workaround #2

This workaround utilizes the equivalent MSUB.Q or MADD.Q instruction that uses the upper half of register D[b]. However there is also an erratum on these instructions (CPU_TC.099), so this workaround takes this into account.

The workaround provides the same result and PSW flags as the original instruction, however it may require an unused data register to be available.

```
MADD.Q D4, D2, D0, D1 L, #1
```

Using just this workaround becomes

```
SH     D7, D1, #16   ; Shift to upper halfword
MADD.Q D4, D2, D0, D7 U, #1
```

combining this workaround with the workaround for CPU_TC.099:

```
SH     D7, D1, #16   ; Shift to upper halfword

MUL.Q  D4, D0, D7 U, #0
JNZ.T  D4, 31, no_bug
JZ.T   D4, 30, no_bug
mac_erratum_condition:
MOVH   D4, #0x8000   ; 0x8000_0000
SUB    D4, D2, D4    ; SUB-1=ADD+1, set V/AV, not C
J      mac_complete
no_bug:
MADD.Q  D4, D2, D0, D7 U, #1
mac_complete:
```

CPU_TC.079 Possible invalid ICR.PIPN when no interrupt pending

Under certain circumstances the Pending Interrupt Priority Number, ICR.PIPN, may be invalid when there is no interrupt currently pending. When no interrupt is pending the ICR.PIPN field is required to be zero.

There are two circumstances where ICR.PIPN may have a non-zero value when no interrupt is pending:

1. When operating in 2:1 mode between CPU and interrupt bus clocks, the ICR.PIPN field may not be reset to zero when an interrupt is acknowledged by the CPU.
2. During the interrupt arbitration process the ICR.PIPN is constructed in 1-4 arbitration rounds where 2 bits of the PIPN are acquired each round. The intermediate PIPN being used to construct the full PIPN is made available as ICR.PIPN. This is a potential problem because reading the PIPN can indicate a pending interrupt that is not actually pending and may not even be valid. e.g. if interrupt 0x81 is the highest priority pending interrupt, then ICR.PIPN will be read as 0x80 during interrupt arbitration rounds 2,3 and 4. Only when the arbitration has completed will the valid PIPN be reflected in ICR.PIPN.

The hardware implementation of the interrupt system for the TriCore1 CPU actually comprises both the PIPN and a separate, non-architecturally visible, interrupt request flag. The CPU only considers PIPN when the interrupt request flag is asserted, at which times the ICR.PIPN will always hold a valid value. As such the hardware implementation of the interrupt priority scheme functions as expected. However, reads of the ICR.PIPN field by software may encounter invalid information and should not be used.

Workaround

None.

CPU_TC.080 No overflow detected by DVINIT instruction for MAX_NEG / -1

A problem exists in variants of the Divide Initialisation instruction with certain corner case operands. Only those instruction variants operating on signed operands, DVINIT, DVINIT.H and DVINIT.B, are affected. The problem occurs when the maximum representable negative value of a number format is divided by -1.

The Divide Initialisation instructions are required to initialise an integer division sequence and detect corner case operands which would lead to an incorrect final result (e.g. division by 0), setting the overflow flag, PSW.V, accordingly.

In the specific case of division of the maximum negative 32-bit signed integer (0x80000000) by -1 (0xFFFFFFFF), the result is greater than the maximum representable positive 32-bit signed integer and should flag overflow. However, this specific case is not detected by the DVINIT instruction and a subsequent division sequence returns the maximum negative number as a result with no corresponding overflow flag.

In the cases of division of the maximum negative 16/8-bit signed integers (0x8000/0x80) by -1 (0xFFFF/0xFF), the result is greater than the maximum representable positive 16/8-bit signed integer and should again flag overflow. These specific cases are not detected by the DVINIT.H/.B instructions with no corresponding overflow flag set. In this case the result of a subsequent division sequence returns the value 0x00008000/0x00000080 which is the correct value when viewed as a 32-bit number but has overflowed the original number format.

Workaround

If the executing program is using the PSW fields to detect overflow conditions, the specific corner case operands described above must be checked for and handled as a special case in software before the standard division sequence is executed.

CPU_TC.081 Error during Load A[10], Call / Exception Sequence

A problem may occur when an address register load instruction, LD.A or LD.DA, targeting the A[10] register, is immediately followed by an operation causing a context switch. The problem may occur in one of two situations:

1. The address register load instruction, targeting A[10], is followed immediately by a call instruction (CALL, CALLA, CALLI).
2. The address register load instruction, targeting A[10], is followed immediately by a context switch caused by an interrupt or trap being taken, where the interrupt stack is already in use (PSW.IS = 1).

In both these situations the value of A[10] is required to be maintained across the context switch. However, where the context switch is preceded by a load to A[10], the address register dependency is not detected correctly and the called

context inherits the wrong value of A[10]. In this case the value of A[10] before the load instruction is inherited.

Example:

```
...
LD.A   A10, <any addressing mode>
CALL   call_target_
...
```

Workaround

The problem only occurs when A[10] is loaded directly from memory. The software workaround therefore consists of loading another address register from memory and moving the contents to A[10].

Example:

```
...
LD.A   A12, <any addressing mode>
MOV.AA A10, A12
CALL   call_target_
...
```

CPU_TC.082 Data corruption possible when Memory Load follows Context Store

Data corruption may occur when a context store operation, STUCX or STLCX, is immediately followed by a memory load operation which reads from the last double-word address written by the context store.

Context store operations store a complete upper or lower context to a 16-word region of memory, aligned on a 16-word boundary. If the context store is immediately followed by a memory load operation which reads from the last double-word of the 16-word context region just written, the dependency is not detected correctly and the previous value held in this memory location may be returned by the memory load.

The memory load instructions which may return corrupt data are as follows:

ld.b, ld.bu, ld.h, ld.hu, ld.q, ld.w, ld.d, ld.a, ld.da

Example:

```
...
STLCX   0xD0000040
LD.W    D15, 0xD0000078
...
```

Note that the TriCore architecture does not require a context save operation (CALL, SVLCX, etc.) to update the CSA list semantically before the next operation (but does require the CSA list to be up to date after the execution of a DSYNC instruction). As such the same problem may occur for context save operations, but the result of such a sequence is architecturally undefined in any case.

Workaround

One NOP instruction must be inserted between the context store operation and a following memory load instruction if the memory load may read from the last double-word of the 16-word context region just written.

Example:

```
...
STLCX   0xD0000040
NOP
LD.W    D15, 0xD0000078
...
```

CPU_TC.083 Interrupt may be taken following DISABLE instruction

The TriCore Architecture requires that the DISABLE instruction gives deterministic behaviour, i.e. no interrupt may be taken following the execution of the DISABLE instruction.

However, the current implementation allows an interrupt to be taken immediately following the execution of the DISABLE instruction, i.e. between the DISABLE and the following instruction. Once the first instruction after the DISABLE instruction has been executed its is still guaranteed that no interrupt will be taken.

Due to this error, when an interrupt is taken **immediately** following a DISABLE instruction, PCXI.PIE will contain the anomalous value 0_B within the interrupt context. In this case, no information is lost, and ICR.IE will be correctly restored upon execution of the corresponding RFE instruction.

Workaround

If an instruction sequence must not be interrupted, then the DISABLE instruction must be followed by a single NOP instruction, before the critical code sequence.

CPU_TC.085 CPS module may error acknowledge valid read transactions

A bug exists in the CPS module, which may result in the CPS incorrectly returning an error acknowledge for a read access to a valid CPS address.

The problem occurs when a read access to a CPS address, in the range 0xF7E00000 - 0xF7E1FFFF, is followed immediately on the FPI bus by a User mode write access to an address with FPI address[16] = 1. The problem occurs due to an error in the FPI bus decoding within the CPS which incorrectly interprets the second transaction, even if to another slave, as an illegal User mode write to a TriCore CSFR and incorrectly error acknowledges the valid read. Write accesses to the CPS module are not affected.

Workaround

For devices in which multiple FPI bus masters may operate in User mode, but only the TriCore CPU and Debug Interface (Cerberus) are expected to access the CPS address range, the workaround consists of 3 parts:

Tool Vendor

1) The Cerberus module must be configured to operate in Supervisor mode, to reduce the probability of the TriCore CPU from receiving false error acknowledges.

2) If the Cerberus FPI Master receives an error acknowledge it enters error state, which is detected by the debugger as a timeout. In this case the debugger should release the Cerberus from the error state with the io_supervisor

command and read out the cause of the error. Where an error acknowledge is determined to be the cause for a read in the CPS address range the read request should be re-issued.

User

3) If the TriCore CPU reads from a CPS address, via the LFI bridge, which results in an error acknowledge being incorrectly generated, the TriCore CPU will take a synchronous DSE trap. In order to workaround this potential problem the following sequence is recommended:

i) A flag is set in a specific memory location immediately before the TriCore CPU attempts a load from a CPS SFR address, and cleared immediately afterwards.

ii) The DSE trap handler is modified to check the status of the flag set in (i). If the flag is set the DSE handler should clear the error capture mechanisms of the FPI BCU and LBCU which will have captured the error acknowledge, and then execute an RFE instruction. This will cause the original load instruction to be re-executed and allow the program to continue normally.

CPU_TC.086 Incorrect Handling of PSW.CDE for CDU trap generation

An error exists in the CDU (Call Depth Underflow) trap generation logic. CDU traps are architecturally defined to occur when "A program attempted to execute a RET (Return) instruction while Call Depth Counting was enabled, and the Call Depth Counter was zero". Call depth counting is enabled when PSW.CDC /= 1111111 and PSW.CDE = 1. However, the status of PSW.CDE is currently not considered for CDU trap generation, and CDU traps may be generated when PSW.CDE = 0.

Call depth counting, and generation of the associated CDO and CDU traps, may be disabled by one of two methods. Setting PSW.CDC = 1111111 globally disables call depth counting and operates as specified. Setting PSW.CDE = 0 temporarily disables call depth counting (it is re-enabled by each call instruction) and is used primarily for call/return tracing.

Workaround

In order to temporarily disable call depth counting for a single return instruction, PSW.CDC should be set to 1111111 before the return instruction is executed.

CPU_TC.087 Exception Prioritisation Incorrect

The TriCore Architecture defines an exception priority order, consisting of the relative priorities of asynchronous traps, synchronous traps and interrupts, and the prioritisation of individual trap types.

The current implementation of the TriCore1 CPU complies with the general principle that the older the instruction is in the instruction sequence which caused the trap, the higher the priority of the trap. However, the relative prioritisation of asynchronous and synchronous events and the prioritisation between individual trap types does not fully comply with the architectural definition.

The current TriCore1 CPU implements the following priority order between an asynchronous trap, a synchronous trap, and an interrupt:

1. Synchronous traps detected in Execute pipeline stage (highest priority).
2. Asynchronous trap.
3. Interrupt.
4. Synchronous trap detected in Decode pipeline stage (lowest priority).

Within these groups the following priorities are implemented:

Table 7 Synchronous Trap Priorities (Detected in Execute Stage)

Priority	Type of Trap
1	VAF-D
2	VAP-D
3	MPR
4	MPW
5	MPP
6	MPN
7	ALN

Table 7 Synchronous Trap Priorities (Detected in Execute Stage)

Priority	Type of Trap
8	MEM
9	DSE
10	OVF
11	SOVF
12	Breakpoint Trap (BAM)

Table 8 Asynchronous Trap Priorities

Priority	Type of Trap
1	NMI
2	DAE

Table 9 Synchronous Trap Priorities (Detected in Decode Stage)

Priority	Type of Trap
1	FCD
2	VAF-P
3	VAP-P
4	PSE
5	Breakpoint Trap (Virtual Address, BBM)
6	Breakpoint Trap (Instruction, BBM)
7	PRIV
8	MPX
9	GRWP
10	IOPC
11	UOPC
12	CDO
13	CDU
14	FCU
15	CSU

Table 9 Synchronous Trap Priorities (Detected in Decode Stage)

Priority	Type of Trap
16	CTYP
17	NEST
18	SYSCALL

Although the implemented trap priorities do not match those defined by the TriCore architecture, this does not cause any problem in the majority of circumstances. The only circumstance in which the incorrect priority order must be considered is in the individual trap handlers, which should not be written to be dependent on the architecturally defined priority order. For instance, according to the architectural definition, a PSE trap handler could assume that any PSE trap received was as a result of a program fetch access from a memory region authorised by the memory protection system. However, as a result of the implemented priorities of PSE and MPX traps, this assumption cannot be made.

Workaround

Trap handlers must be written to take account of the implemented priority and not rely upon the architecturally defined priority order.

CPU_TC.088 Imprecise Return Address for FCU Trap

The FCU trap is taken when a context save operation is attempted but the free context list is found to be empty, or when an error is encountered during a context save or restore operation. In failing to complete the context operation, architectural state is lost, so the occurrence of an FCU trap is a non-recoverable system error.

Since FCU traps are non-recoverable system errors, having a precise return address is not important, but can be useful in establishing the cause of the FCU trap. The current TriCore1 implementation does not generate a precise return address for FCU traps in all circumstances.

An FCU trap may be generated as a result of 3 situations:

1. An instruction caused a context operation explicitly (CALL, RET etc.), which failed. The FCU return address should point to the instruction which caused the context operation.
2. An instruction caused a synchronous trap, which attempted to save context and encountered an error. The FCU return address should point to the original instruction which caused the synchronous trap.
3. An asynchronous trap or interrupt occurred, which attempted to save context and encountered an error. The FCU return address should point to the next instruction to be executed following a return from the asynchronous event.

In each of these circumstances the return address generated by the current TriCore1 implementation may be up to 8 bytes greater than that intended.

Workaround

None

CPU_TC.089 Interrupt Enable status lost when taking Breakpoint Trap

The Breakpoint Trap allows entry to a Debug Monitor without using user resources, irrespective of whether interrupts are enabled or not.

Early revisions of the TriCore Architecture manual, up to and including version V1.3.5, state that the actions pertaining to the `ICR.IE` bit upon taking a breakpoint trap are:

- Write PCXI to DCX + 0_H.
- `ICR.IE` = 0_H.

Upon returning from a breakpoint trap, the corresponding action taken is:

- Restore PCXI from DCX + 0_H.

Unfortunately, during such a breakpoint trap, return from monitor sequence the original status of the interrupt enable bit, `ICR.IE`, is lost. `ICR.IE` is cleared to disable interrupts by the breakpoint trap, but the previous value of `ICR.IE` is not stored. The desired behaviour is to store `ICR.IE` to `PCXI.PIE` on taking a breakpoint trap, and restore it upon return from the debug monitor. The current TriCore1 implementation matches the early architecture definition whereby the interrupt enable status is lost on taking a breakpoint trap.

Workaround

If breakpoint traps are used in conjunction with code where the original status of the `ICR.IE` bit is known, then the debug monitor may set `ICR.IE` to the desired value before executing the return from monitor.

If the original status of `ICR.IE` is not known and cannot be predicted, an alternative debug method must be used, such as an external debugger or breakpoint interrupts.

CPU_TC.090 MMU Page Table Entry Mapping Restrictions

The TriCore V1.3 architecture defines a number of restrictions regarding Page Table Entries (PTEs) which should not be installed in the MMU (using the TLBMAP instruction).In addition to these documented restrictions, the current TriCore1.3 implementation imposes further restrictions on PTEs that should not be installed. Installing a PTE in contravention of these restrictions will result in undefined behaviour.

General restrictions are as follows:

- A PTE must not contain a VPN where the virtual address is in the upper half of the address space.
- A PTE must not contain a PPN where the physical address is in a peripheral segment (segment E or F).
- A PTE where the physical address obtained from the PPN is in a non-cacheable memory region must not have the PTE Cacheability bit (C) set.

Where the physical address obtained from the PPN is in a cacheable memory region and the PTE Cacheability bit (C) is set, additional restrictions are imposed as follows:

- For a 4KByte cache, either a page size greater than 1KByte must be used, or VPN[0] must match PPN[0].
- For an 8KByte cache, either a page size greater than 1KByte must be used, or VPN[1:0] must match PPN[1:0].
- For a 16KByte cache, either a page size greater than 4KByte must be used, or VPN[2:0] must match PPN[2:0] (assuming 1KByte page size).

For example, the TC1130 device has a 16KByte program cache and a 4KByte data cache. Any PTE used exclusively for data accesses (PTE.XE = 0) must

comply with the restriction for a 4K cache, whilst any PTE used for program access must comply with the restriction for a 16KByte cache.

The MMU may also be used to map virtual addresses to physical addresses which are in the range of the data and program scratchpad memories. In this case a further restriction applies as follows:

- Either a page size greater than the scratchpad memory size must be used, or for those address bits used to access the scratchpad memory, the corresponding VPN bits must equal the PPN bits.

example

the TC1130 device contains 32KByte Program Scratchpad RAM (PSPR) and address bits [14:0] are used to access a location within this memory. For a 1KByte page size, the VPN and PPN contain 22 bits, with VPN/PPN[21:0] mapping to address bits [31:10]. In order to access the program scratchpad RAM via a PTE-based translation using a 1KByte page size, VPN[4:0] (address [14:10]) must equal PPN[4:0].

CPU_TC.091 Incorrect privilege handling of MMU instructions

The TriCore V1.3 architecture defines the MMU instructions (TLBMAP, TLBDEMAP etc.) to be privileged instructions, executable in Supervisor mode only. Any attempt to execute an MMU instruction in a User mode should result in a PRIV trap.

However, the current TriCore1.3 implementation allows the MMU instructions to be executed in User-1 mode. Any attempt to execute an MMU instruction in User-0 mode will result in an MPP trap

Workaround

None.

CPU_TC.092 Upper Memory Segments accessible in User-0 Mode with MMU enabled

The TriCore V1.3 architecture defines that for any system with an MMU, which is operating in virtual mode (MMU_CON.V = 1), then any User-0 mode access to a virtual address in the upper segments (which is not a peripheral segment) should result in a VAP trap.

The current TriCore1.3 implementation does not enforce this restriction and accesses to such upper memory segments in User-0 mode, with the TriCore operating in virtual mode, will be permitted.

Workaround

In order to prevent User-0 mode tasks from accessing the upper memory segments directly, the range-based memory protection system should be used to enforce the required behaviour.

CPU_TC.093 MMU Instruction Usage Restrictions

The TriCore Memory Management Unit (MMU) contains arbitration logic to handle the situation where multiple requests to access the UTLB occur concurrently, by instruction fetches, load-store instructions and/or MMU instructions. In the case of concurrent instruction fetch and load-store instruction accesses, this arbitration logic operates as required. However, when MMU instructions (TLBMAP, TLBDEMAP, etc.) require access to the MMU UTLB concurrent with either instruction fetch or load-store instruction accesses, the UTLB arbitration logic can fail and give undefined results.

Workaround

In order to avoid the problems in the UTLB arbitration logic, any MMU instruction, which is not followed by another MMU instruction, must be followed by a NOP and an ISYNC instruction. Multiple MMU instructions may be executed back-to-back without the need for intermediate NOP+ISYNC. In addition, all MMU instructions should be executed from addresses undergoing direct translation, such that instruction fetches do not require the UTLB.

Example:

```
...
TLBMAP E0
TLBMAP E2
NOP
ISYNC
...
```

CPU_TC.094 Potential Performance Loss when CSA Instruction follows IP Jump

The TriCore1 CPU contains shadow registers for the upper context registers, to optimise the latency of certain CSA list operations. As such, the latency of instructions operating on the CSA list is variable dependent on the state of the context system. For instance, a return instruction will take fewer cycles when the previous upper context is held in the shadow registers than when the shadow registers are empty and the upper context has to be re-loaded from memory.

In situations where the CSA list is located in single cycle access memory (i.e. Data Scratchpad RAM), instructions operating on the upper context (such as call, return) will have a latency of between 2 and 5 cycles, dependent on the state of the context system. In the case where the CSA list instruction will take 4 or 5 cycles, the instruction will cause the instruction fetch request to be negated whilst the initial accesses of the context operation complete.

A performance problem exists when certain jump instructions which are executed by the integer pipeline are followed immediately by certain CSA list instructions, such that the instructions are dual-issued. In this case, where the jump instruction is predicted taken, the effect of the CSA list instruction on the fetch request is not immediately cancelled, which can lead to the jump instruction taking 2 cycles longer than expected. This effect is especially noticeable where the jump instruction is used to implement a short loop, since the loop may take 2 cycles more than expected. In addition, since the state of the context system may be modified by asynchronous events such as interrupts, the execution time of the loop before and after an interrupt is taken may be different.

Integer pipeline jump instructions are those that operate on data register values as follows:

JEQ, JGE, JGE.U, JGEZ, JGTZ, JLEZ, JLT, JLT.U, JLTZ, JNE, JNED, JNEI, JNZ, JNZ.T, JZ, JZ.T

CSA list instructions which may cause the performance loss are as follows:

CALL, CALLA, CALLI, SYSCALL, RET, RFE

Workaround

In order to avoid any performance loss, in particular where the IP jump instruction is used to implement a loop and as such is taken multiple times, a NOP instruction should be inserted between the IP jump and the CSA list instruction.

Example:

```
...
JLT.U  D[a], D[b], jump_target_
NOP
RET
...
```

CPU_TC.095 Incorrect Forwarding in SAT, Mixed Register Instruction Sequence

In a small number of very specific instruction sequences, involving Load-Store (LS) pipeline instructions with data general purpose register (DGPR) operands, the operand forwarding in the TriCore1 CPU may fail and the data dependency between two instructions be missed, leading to incorrect operation. The problem may occur in one of two instruction sequences as follows:

Problem Sequence 1)

1. LS instruction with DGPR destination {mov.d, eq.a, ne.a, lt.a, ge.a, eqz.a, nez.a, mfcr}
2. SAT.H instruction
3. LS instruction with DGPR source {addsc.a, addsc.at, mov.a, mtcr}

If the DGPR source register of (3) is equal to the DGPR destination register of (1), then the interaction with the SAT.H instruction may cause the dependency to be missed and the original DGPR value to be passed to (3).

Problem Sequence 2)

1. Load instruction with 64-bit DGPR destination {ld.d, ldlcx, lducx, rslcx, rfe, rfm, ret}
2. SAT.B or SAT.H instruction
3. LS instruction with DGPR source {addsc.a, addsc.at, mov.a, mtcr}

In this case if the DGPR source register of (3) is equal to the high 32-bit DGPR destination register of (1), then the interaction with the SAT.B/SAT.H instruction may cause the dependency to be missed and the original DGPR value to be passed to (3).

Example:

```
...
MOV.D   D2, A12
SAT.H   D7
MOV.A   A4, D2
...
```

Note that for the second problem sequence the first instruction of the sequence could be RFE and as such occur asynchronous with respect to the program flow.

Workaround

A single NOP instruction must be inserted between any SAT.B/SAT.H instruction and a following Load-Store instruction with a DGPR source operand {addsc.a, addsc.at, mov.a, mtcr}.

CPU_TC.096 Error when Conditional Loop targets Single Issue Group Loop

An error in the program flow may occur when a conditional loop instruction (LOOP) has as its target an instruction which forms part of a single issue group loop. Single issue group loops consist of an optional Integer Pipeline (IP) instruction, optional Load-Store Pipeline (LS) instruction and a loop instruction targeting the first instruction of the group. In order for the problem to occur the outer loop must first be cancelled (for instance due to a pipeline hazard) before being executed normally. When the problem occurs the loop counter of the outer loop instruction is not decremented correctly and the loop executed an incorrect number of times.

Example:

```
...
loop_target_:
ADD    D2, D1          ; Optional IP instruction
ADD.A  A2, A1          : Optional LS instruction
LOOP   Ax, loop_target_; Single Issue Group Loop
...
LD.A   Am, <addressing mode>
LD.W   Dx, [Am]        ; Address dependency causes cancel
LOOP   Ay, loop_target_; Conditional loop targets
                       ; single issue group loop
...
```

Workaround

Single issue group loops should not be used. Where a single issue group loop consists of an IP instruction and a loop instruction targeting the IP instruction, two NOPs must be inserted between the IP and loop instructions. Where a single issue group loop consists of an optional IP instruction, a single LS instruction and a loop instruction targeting the first instruction of this group, a single NOP must be inserted between the LS instruction and the loop instruction. Since single issue group loops do not operate optimally on the current TriCore1 implementation (not zero overhead), no loss of performance is incurred.

CPU_TC.097 Overflow wrong for some Rounding Packed Multiply-Accumulate instructions.

An error is made in the computation of the overflow flag (PSW.V) for some of the rounding packed multiply-accumulate (MAC) instructions. The error affects the following instructions with a 64bit accumulater input:

MADDR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=43_H

MSUBR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=63_H

PSW.V is computed by combining ov_halfword1 and ov_halfword0, as described in the TriCore architecture manual (V1.3.6 and later) for these instructions. When the error conditions exist ov_halfword1 is incorrectly computed. ov_halfword0 is always computed correctly.

Note: Under the error conditions, PSW.V may be correct depending on the value of ov_halfword0.

The specific error conditions are complex and are not described here.

Workaround #1

If the PSW.V and PSW.SV flags generated by these instructions are not used by the code, then the instructions can be used without a workaround.

Workaround #2

If the algorithm allows use of 16 bit addition inputs, the code could be rewritten to use the following instructions instead:

MADDR.H D[c], **D**[d], D[a], D[b] UL, n; opcode[23:18]=$0C_H$, opcode[7:0]=83_H

MSUBR.H D[c], **D**[d], D[a], D[b] UL, n; opcode[23:18]=$0C_H$, opcode[7:0]=$A3_H$

Workaround #3

If the PSW.V and PSW.SV flags are used, and 32 bit addition inputs are required, then the routine should be rewritten to use two unpacked mac instructions. I.e.

```
MADDR.H     D4, E2, D0, D1 UL, #n
```

Becomes

```
MADDR.Q     D4, D3, D0 U, D1 U, #n
```

```
MADDR.Q       D5, D2, D0 L, D1 L, #n
SH            D5, D5, #-16
INSERT        D4, D4, D5, #16, #16; Repack into D4
```

Note: PSW.V must be tested between the two MADDR.Q instructions if PSW.SV cannot be utilised.

Note: This algorithm requires an additional register (D5 in the example).

Workaround #3 for erroneous MSUBR.H instruction is similar to the MADDR.H instruction.

CPU_TC.098 Possible PSW.V Error for an MSUB.Q instruction variant when both multiplier inputs are of the form 0x8000xxxx

The bug only affects the following instruction

MSUB.Q D[c], D[d], D[a], D[b] , n; opcode[23:18]=02_H, opcode[7:0]=63_H

PSW.V is computed by the algorithm in the TriCore Architecture Manual for this instruction except under the following conditions:

(D[a][31:16] = 16'h8000) &&

(D[b][31:16] = 16'h8000) &&

(n = 1)

When these conditions are met the following algorithm is used to produce the incorrect PSW.V

```
if expected (PSW.V) = 1     // expected to overflow
   PSW.V = 0
 else                       // not expected to overflow
   if (result < 0) and (D[d] >= 0)
     PSW.V = 1
   else
     PSW.V = 0
   endif
 endif
```

Workaround #1

If the PSW.V and PSW.SV flags generated by this instruction are not used by the code, then the instruction can be used without a workaround.

Workaround #2

Use the equivalent instruction which produces a 64 bit result.

MSUB.Q E[c], E[d], D[a], D[b] , n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

To use the 64 bit version, D[d] should occupy the odd word of E[d], the even word of E[d] should be set to zero. The result will appear in the odd word of E[c].

Note: This version of the MSUB.Q instruction is affected by another erratum CPU_TC.099. Please ensure that the workaround for that erratum is implemented.

This workaround provides the same result and PSW flags as the original instruction, however it requires additional unused data registers to be available.

CPU_TC.099 Saturated Result and PSW.V can error for some q format multiply-accumulate instructions when computing multiplications of the type 0x80000000*0x8000 when n=1

For some q format multiply-accumulate instructions, the overflow flag (PSW.V) is computed incorrectly under some circumstances. When the problem behaviour occurs, the overflow flag is always generated incorrectly: if PSW.V should be set it will be cleared, and if it should be cleared it will be set.

Where this bug affects a saturating instruction the result is incorrectly saturated.

This bug affects the following instructions:

32bit * 32bit Instructions

MUL.Q D[c], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=93_H

MUL.Q E[c], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=93_H

MADD.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=43_H

MADD.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=43_H

MSUB.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

32bit * 16bit (Upper) Instructions

MUL.Q D[c], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=93_H

MADD.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=43_H

MADDS.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=20_H, opcode[7:0]=43_H

MSUB.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=63_H

MSUBS.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=20_H, opcode[7:0]=63_H

32bit * 16bit (Lower) Instructions

MUL.Q D[c], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=93_H

MADDS.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=21_H, opcode[7:0]=43_H

MSUBS.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=21_H, opcode[7:0]=63_H

The error condition occurs, and hence PSW.V is inverted under the following conditions:

32bit * 32bit Instructions

D[a] = 32’h8000_0000 and

D[b] = 32’h8000_0000 and

n = 1

32bit * 16bit (Upper) Instructions

D[a] = 32’h8000_0000 and

D[b][31:16] = 16’h8000 and

n = 1

32bit * 16bit (Lower) Instructions

D[a] = 32’h8000_0000 and

D[b][15:0] = 16’h8000 and

n = 1

When the error condition occurs for a saturating instruction, the result is wrong in addition to PSW.V. The result in these cases is as follows:

MADDS.Q, PSW.V incorrectly asserted

32 bit result: D[c] = 32'h8000_0000

MADDS.Q, PSW.V incorrectly negated

32 bit result: D[c] = result[31:0]

MSUBS.Q, PSW.V incorrectly asserted

32 bit result: D[c] = 32'h7FFF_FFFF

MSUBS.Q, PSW.V incorrectly negated

32 bit result: D[c] = result[31:0]

Workaround #1

For instructions which don't saturate, if the PSW.V and PSW.SV flags generated by the instruction are not used by the code, then the instruction can be used without a workaround.

Workaround #2

Prior to executing the erroneous instruction, test the operands to detect the error condition. If the error condition exists, execute an alternative routine. Detecting the error condition is performed by executing a MUL.Q on the multiplicands with n=0, then testing bit 30 of the result which is only set when the error condition operands exist.

Each erroneous instruction can be replaced by the relevant code sequence described below.

Note: If the destination register is the same as one of the source registers, then an additional data register will be needed to implement the workaround.

MUL.Q D[c], D[a], D[b], #1; opcode[23:18]=02_H, opcode[7:0]=93_H

```
MUL.Q   D4, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x4000 ; 0x4000_0000
  ADD     D4, D4, D4  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1, #1
mac_complete:
```

MUL.Q E[c], D[a], D[b], #1; opcode[23:18]=1B$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   E4, D0, D1, #1
```

becomes

```
  MUL.Q   E4, D0, D1, #0
  JNZ.T   D5, 31, no_bug
  JZ.T    D5, 30, no_bug
mac_erratum_condition:
  MOV     D4, #0
  MOVH    D5, #0x4000 ; 0x4000_0000
  ADD     D5, D5, D5  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   E4, D0, D1, #1
mac_complete:
```

MUL.Q D[c], D[a], D[b] U, #1; opcode[23:18]=00$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   D4, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x4000 ; 0x4000_0000
```

```
  ADD     D4, D4, D4  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1 U, #1
mac_complete:
```

MUL.Q D[c], D[a], D[b] L, #1; opcode[23:18]=01_H, opcode[7:0]=93_H

```
  MUL.Q   D4, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #4000   ; 0x4000_0000
  ADD     D4, D4, D4  ; 0x8000_0000 set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1 L, #1
mac_complete:
```

MADD.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=02_H, opcode[7:0]=43_H

```
  MADD.Q  D4, D2, D0, D1 #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUB     D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  D4, D2, D0, D1, #1
mac_complete:
```

MADD.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=1B$_H$, opcode[7:0]=43$_H$

```
  MADD.Q  E4, E2, D0, D1 #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOV     D4, D2      ; lower word add 0
  MOVH    D5, #0x8000 ; 0x8000_0000
  SUB     D5, D3, D5  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  E4, E2, D0, D1, #1
mac_complete:
```

MADD.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=00$_H$, opcode[7:0]=43$_H$

```
  MADD.Q  D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUB     D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  D4, D2, D0, D1 U, #1
mac_complete:
```

MADDS.Q D[c], D[d], D[a], D[b]U, #1; opcode[23:18]=20$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
```

```
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUBS    D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADDS.Q D4, D2, D0, D1 U, #1
mac_complete:
```

MADDS.Q D[c], D[d], D[a], D[b] L, #1; opcode[23:18]=21_H, opcode[7:0]=43_H

```
  MADDS.Q D4, D2, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, #31, no_bug
  JZ.T    D4, #30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUBS    D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADDS.Q D4, D2, D0, D1 L, #1
mac_complete:
```

MSUB.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

```
  MSUB.Q  E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOV     D4, D2      ; lower word add 0
  MOVH    D5, #0x8000 ; 0x8000_0000
  ADD     D5, D3, D5  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
```

```
no_bug:
  MSUB.Q  E4, E2, D0, D1, #1
mac_complete:
```

MSUB.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=00_H, opcode[7:0]=63_H

```
  MSUB.Q  D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADD     D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUB.Q  D4, D2, D0, D1 U, #1
mac_complete:
```

MSUBS.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=20_H, opcode[7:0]=63_H

```
  MSUBS.Q D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADDS    D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUBS.Q D4, D2, D0, D1 U, #1
mac_complete:
```

MSUBS.Q D[c], D[d], D[a], D[b] L, #1; opcode[23:18]=21_H, opcode[7:0]=63_H

```
  MSUBS.Q D4, D2, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADDS    D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUBS.Q D4, D2, D0, D1 L, #1
mac_complete:
```

CPU_TC.100 Mac instructions can saturate the wrong way and have problems computing PSW.V

Under certain error conditions, some saturating mac instructions saturate the wrong way. I.e. if they should saturate to the maximum positive representable number, they saturate to the maximum negative representable number, and vice versa.

In addition to this problem, the affected instructions also compute the overflow flag (PSW.V) incorrectly under certain circumstances. If PSW.V should be set it will be cleared, and if it should be cleared it will be set. When PSW.V is wrong, the instructions' results are wrong due to incorrect saturation.

The following instructions are subject to these errors:

MADDS.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=22_H, opcode[7:0]=43_H

MADDS.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$3B_H$, opcode[7:0]=43_H

MSUBS.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=22_H, opcode[7:0]=63_H

MSUBS.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$3B_H$, opcode[7:0]=63_H

The PSW.V is computed incorrectly under the following circumstances:

```
D[a] = 32'h8000_0000 and
D[b] = 32'h8000_0000 and
n = 1
```

Note: When n=0 all affected instructions operate correctly.

Workaround #1

Use the non saturating version of the instruction if the algorithm allows its use.

MADD.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=43_H

MADD.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=43_H

MSUB.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

Note: These alternative instructions are subject to erratum CPU_TC.0.99. Please ensure that the workaround for that erratum is implemented.

MSUB.Q D[c], D[d], D[a], D[b] , n; opcode[23:18]=02_H, opcode[7:0]=63_H

Note: This alternative instruction is subject to erratum CPU_TC.098. Please ensure that the workaround for that erratum is implemented.

Workaround #2

Prior to executing the erroneous instruction, test the operands to detect the PSW.V error condition. If the error condition exists, execute an alternative routine.

Following this routine PSW.V will be correct, but the result may have saturated incorrectly. So now determine which way the instruction should have saturated (if at all) and saturate manually.

Each erroneous instruction can be replaced by the relevant code sequence described below.

Note: An additional data register is needed to implement this workaround.

Note: The PSW.USB are destroyed by this workaround.

MADDS.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=22_H, opcode[7:0]=43_H

```
  MADDS.Q D4, D2, D0, D1, #1
```

becomes

```
  ; First correct the PSW.V problem
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
```

```
  MOVH    D4, #0x8000           ; 0x8000_0000
  SUBS    D4, D2, D4            ; SUB -1 == ADD +1
  J       mac_complete          ; Saturation correct
no_v_bug:
  MADDS.Q D4, D2, D0, D1, #1
  ; PSW.V correct, but res may have saturated wrong way
  MFCR    D7, #0xFE04           ; get PSW
  JZ.T    D7, 30, mac_complete  ; End if no sat required
saturate:
  MOVH    D4, #0x8000           ; 0x80000000
  XOR     D7, D0, D1            ; Test sign of mul output
                                ; +ve => sat to max
  JNZ.T   D7, 31, mac_complete  ; if sat to min, finish
saturate_max:
  MOV     D7, #-1
  ADD     D4, D4, D7          ; 0x80000000 -1 = 0x7fffffff
mac_complete:
```

MADDS.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=3B$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOV     D4, D2                ; Lower word not modified
  ; Compute Upper Word
  MOVH    D5, #0x8000           ; 0x8000_0000
  SUB     D5, D3, D5            ; SUB -1 == ADD +1, set V
  J       test_v                ; perform sat64
no_v_bug:
  MADDS.Q E4, E2, D0, D1, #1
test_v:
  ; PSW.V correct, res may have saturated the wrong way
  MFCR    D7, #0xFE04           ; get PSW
  JZ.T    D7, 30, mac_complete  ; End if no sat required
```

```
saturate:
  MOVH    D5, #0x8000           ; 0x80000000_00000000
  MOV     D4, #0
  XOR     D7, D0, D1            ; Test sign of mul output
                                ; +ve => sat to max
  JNZ.T   D7, 31, mac_complete  ; if sat to min, finish
saturate_max:
  MOV     D4, #-1
; 0x80000000_00000000 -1 = 0x7fffffff_ffffffff
  ADD     D5, D5, D4
mac_complete:
```

M**SUBS.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=22_H, opcode[7:0]=63_H**

```
  MSUBS.Q D4, D2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOVH    D4, #0x8000           ; 0x8000_0000
  ADDS    D4, D2, D4            ; ADD -1 == SUB +1
  J       mac_complete          ; Saturation correct
no_v_bug:
  MSUBS.Q D4, D2, D0, D1, #1
  ; Now PSW.V is correct, but result may have saturated the
wrong way
  MFCR    D7, #0xFE04           ; get PSW
  JZ.T    D7, #30, mac_complete ; End no sat required
saturate:
  MOVH    D4, #0x8000           ; 0x80000000
  XOR     D7, D0, D1            ; Test sign of mul output
                                ; -ve => sat to max
  JZ.T    D7, #31, mac_complete ; if sat to min, finish
saturate_max:
  MOV     D7, #-1
  ADD     D4, D4, D7            ; 0x80000000-1=0x7fffffff
```

```
mac_complete:
```

MSUBS.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=3B$_H$, opcode[7:0]=63$_H$

```
  MSUBS.Q E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOV     D4, D2                ; Lower word not modified
  ; Compute Upper Word
  MOVH    D5, #0x8000           ; 0x8000_0000
  ADD     D5, D3, D5            ; ADD -1 == SUB +1, set V
  J       test_v                ; perform sat64
no_v_bug:
  MSUBS.Q E4, E2, D0, D1, #1
  ; Now PSW.V is correct, but result may have saturated the
wrong way
test_v:
  MFCR    D7, #0xFE04           ; get PSW
  JZ.T    D7, #30, mac_complete ; Test V, finish if no
saturation required
saturate:
  MOVH    D5, #0x8000           ; 0x80000000_00000000
  MOV     D4, #0
  XOR     D7, D0, D1            ; Test sign of mul output
                                ; -ve => sat to max
  JZ.T    D7, #31, mac_complete ; if sat to min, finish
saturate_max:
  MOV     D4, #-1
; 0x80000000_00000000 -1 = 0x7fffffff_ffffffff
  ADD     D5, D5, D4
mac_complete:
```

Workaround #3

Where the use of one of these instructions is unavoidable, and both the correct result and PSW.USB are required, the UPDFL instruction can be used to modify PSW.USB in user mode. Note that the UPDFL instruction is only available in systems which have an FPU coprocessor present. The correct result can be obtained by using workaround #2.

CPU_TC.101 MSUBS.U can fail to saturate result, and MSUB(S).U can fail to assert PSW.V

Under certain circumstances two variants of the MSUB.U instruction can fail to assert PSW.V when expected to do so. When this occurs for MSUBS.U, the result fails to saturate.

The error affects the following instructions:

MSUB.U E[c], E[d], D[a], D[b]; opcode[23:18]=68_H, opcode[7:0]=23_H

MSUBS.U E[c], E[d], D[a], D[b]; opcode[23:18]=$E8_H$, opcode[7:0]=23_H

The error exists when the conditions below exist. Note that 'result' is as defined in the architecture manual. Note that D[a][31:16] and D[b][31:16] are both treated as unsigned.

```
(result < 0) and; PSW.V is expected to be asserted
(E[d][63] = 1) and
((D[a][31:16] * D[b][31:16])[31] = 0)
```

When the error conditions exist, PSW.V should be asserted, but is erroneously negated.

For the saturating instruction MSUBS.U, when the error condition exists the returned result (E[c]) is also wrong. Instead of saturating to 0, the return result is as given below:

```
E[c] = result[63:0]
```

Workaround #1

If it can be guaranteed that E[c][63] = 0 under all code execution conditions, then both of these erroneous instructions will produce the correct result and PSW and can therefore be used.

Workaround #2

For MSUB.U, if the PSW.V and PSW.SV flags generated are not used by the code, then the instruction can be used without a workaround.

Workaround #3

For MSUBS.U, if none of the PSW.USB flags are used by the code, then the following workaround can be used to produce the correct saturated result.

Note: This workaround destroys PSW.C

Note: This workaround requires at least one additional data register to be used (D7 in the example), and maybe more, if the destination register is the same as one of the source registers.

```
  MSUBS.U  E4, E2, D0, D1
```

becomes

```
  ; Different routines if PSW.SV set at start
  MUL.U   E4, D0, D1            ; execute mul
  SUBX    D4, D2, D4            ; sub lower word
  SUBC    D5, D3, D5            ; sub upper word
  MFCR    D7, #0xFE04           ; get PSW
  JNZ.T   D7, 31, mac_complete  ; Test PSW.C, no overflow
if set so finish

  ; MSUBS.U overflows, so saturate to zero
  MOV    D4, #0
  MOV    D5, #0
mac_complete:
```

Workaround #4

Where the use of one of these instructions is unavoidable, and both the correct result and PSW.USB are required, the UPDFL instruction can be used to modify PSW.USB in user mode. Note that the UPDFL instruction is only available in systems which have an FPU coprocessor present. The correct result can be obtained by using workaround #3 for MSUBS.U.

CPU_TC.102 Result and PSW.V can be wrong for some rounding, packed, saturating, MAC instructions.

An error is made in the computation of the result and overflow flag (PSW.V) for some of the rounding packed saturating multiply-accumulate (MAC) instructions. The error affects the following instructions with a 64bit accumulater input:

MADDRS.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$3E_H$, opcode[7:0]=43_H

MSUBRS.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$3E_H$, opcode[7:0]=63_H

When these instructions erroneously detect overflow, the results are saturated and PSW.V and PSW.SV are asserted.

PSW.V is computed by combining ov_halfword1 and ov_halfword0, as described in the TriCore Architecture Manual (V1.3.6 and later) for these instructions. When the error conditions exist ov_halfword1 is incorrectly computed. ov_halfword0 is always computed correctly.

Note: Under the error conditions, PSW.V may be correct depending on the value of ov_halfword0.

The specific error conditions are complex and are not described here.

Workaround #1

If the saturating version of the instruction does not need to be used, then consider using the unsaturating versions:

MADDR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=43_H

MSUBR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=63_H

Note: Whilst these instructions compute the result correctly, PSW.V and PSW.SV are still affected by the problem as described in erratum CPU_TC_0.97.

Workaround #2

If the algorithm allows use of 16 bit addition inputs, the code could be rewritten to use the following instructions instead:

MADDRS.H D[c], **D[d]**, D[a], D[b] UL, n; opcode[23:18]=$2C_H$, opcode[7:0]=83_H

MSUBRS.H D[c], **D[d]**, D[a], D[b] UL, n; opcode[23:18]=$2C_H$, opcode[7:0]=$A3_H$

Workaround #3

If the PSW.V and PSW.SV flags are used, and 32 bit addition inputs are required, then the routine should be rewritten to use two unpacked mac instructions. I.e.

```
MADDRS.H      D4, E2, D0, D1 UL, #n
```

Becomes

```
MADDRS.Q      D4, D3, D0 U, D1 U, #n
MADDRS.Q      D5, D2, D0 L, D1 L, #n
SH            D5, D5, #-16
INSERT        D4, D4, D5, #16, #16; Repack results into D4
```

Note: PSW.V must be tested between the two MADDR.Q instructions if PSW.SV cannot be utilised.

Note: This algorithm requires an additional register (D5 in the example)

The workaround for MSUBRS.H instruction is similar to the MADDRS.H instruction.

CPU_TC.103 Spurious parity errors can be generated

Under certain conditions, spurious memory parity errors may be generated by valid memory accesses. Such parity errors typically result in the generation of an NMI trap to the CPU.

Due to the TriCore1 pipeline, and the potential presence of an MMU re-mapping addresses, instruction fetch requests are initially predicted to be to either SPRAM or ICache. The exact target of an instruction fetch is not known until the cycle after the initial memory access. Note that this prediction mechanism operates independently of whether or not an MMU is actually present. In the case of PMEM the spurious parity error problem may occur when an instruction fetch request targeting an SPRAM address is immediately followed by a second instruction fetch request targeting a non-SPRAM address. In this case the second memory access is predicted to target an SPRAM address and the PMEM is speculatively accessed accordingly. In the following cycle the speculative access is cancelled and re-issued to the correct target, but the

speculative access cancellation is not communicated to the parity detection logic. It may happen that the second access, were it targeting SPRAM, maps to a non-valid location in memory hence potentially causing a spurious parity error.

PMEM Workaround

In order to avoid spurious parity errors in the PMEM case, it must be ensured that any instruction fetch to an SPRAM address is not followed immediately by an instruction fetch to a non-SPRAM address with an address which could cause this problem. Any software routine placed in SPRAM should not return directly to the calling code, rather it must jump to a sub routine placed in a 'safe' non-SPRAM address which itself then returns to the calling code.

'Safe' non-SPRAM addresses are defined as any address except:

bit [15:14] = 11_b (TC1130, TC1115, TC1110 PMEM);

The interrupt vector table (or trap vector table), if located in a non-SPRAM location, must also be located at a 'safe' address. If they are located in SPRAM, the return to calling code must be indirect i.e. jump to a safe address and execute the RET/RFE instruction from the 'safe' address.

If this workaround is not possible for a given application, then PMEM parity detection and flagging must be disabled by setting SCU_PETCR.PEN1 = 0.

CPU_TC.104 Double-word Load instructions using Circular Addressing mode can produce unreliable results

Under certain conditions, a double-word load instruction (LD.D) using circular addressing mode can produce unreliable results. The problem occurs when the following conditions are met:

- The effective address of the LD.D instruction using circular addressing mode (Base+Index) is only half-word aligned (not word or double-word aligned) and targets a circular buffer placed in Data Scratchpad RAM (DSPR or LDRAM) or cacheable data memory (where an enabled Data Cache is present).
- The effective address of the LD.D instruction is such that the memory access runs off the end of the circular buffer, with the first three half-words

of the required data at the end of the buffer and last half-word wrapped around to the start of the buffer.

- The TriCore CPU store buffer contains a pending store instruction targeting at least one of the three data half-words from the end of the circular buffer being read.

Note: The TriCore1 CPU contains a single store buffer. A store operation is placed in the store buffer when it is followed in the Load-Store pipeline by a load operation. The store buffer empties when the next store operation occurs or when the Load-Store pipeline contains no memory access operation.

When these conditions are met, the first memory access (to the upper three half-words of the buffer) of the LD.D instruction is made, but the dependency to the pending store instruction is then detected and the access cancelled. The store is then performed in the next cycle and the first access of the LD.D instruction subsequently re-issued. However, in this specific set of circumstances the first access of the LD.D instruction is re-issued incorrectly using the data size of the second access (half-word). As such not all the required data half-words are read from memory.

Under most circumstances this problem is not detectable, since the SRAM memories used hold the previous values read with the data merged from the store operation. However, if another bus master accesses the Data Scratchpad RAM within this sequence, but before the LD.D is re-issued, the SRAM memory outputs no longer default to the required data and the data returned by the LD.D instruction is incorrect.

Example 1:

```
a12 = 0xd0001020
a13 = 0x00180012
...
ST.Q [a12/a13+c]0, d14
LD.D e10, [a12/a13+c]2
...
```

Example 2:

```
a12 = 0xd0001020
a13 = 0x00180012
```

```
...
ST.Q [a12/a13+c]0, d14
LD.W d2, [a4]; Previous ST.Q -> Store Buf
LD.D e10, [a12/a13+c]2 ; ST.Q still in Store Buf
...
```

Workaround

Wherever possible, double-word load instructions using circular addressing mode should be constrained such that their effective address (Base+Index) is word aligned.

Where this is not possible, and where it cannot be guaranteed that the CPU store buffer will not contain an outstanding store operation which could conflict with the LD.D instruction as described previously, the LD.D instruction must be preceded by a NOP.

```
...
ST.Q [a12/a13+c]0, d14
NOP
LD.D e10, [a12/a13+c]2
...
```

CPU_TC.105 User / Supervisor mode not staged correctly for Store Instructions

Bus transactions initiated by TriCore load or store instructions have a number of associated attributes such as address, data size etc. derived from the load or store instruction itself. In addition, bus transactions also have an IO privilege level status flag (User/Supervisor mode) derived from the PSW.IO bit field. Unlike attributes derived from the instruction, the User/Supervisor mode status of TriCore initiated bus transactions is not staged correctly in the TriCore pipeline and is derived directly from the PSW.IO bit field.

This issue can only cause a problem in certain circumstances, specifically when a store transaction is outstanding (e.g. held in the CPU store buffer) and the PSW is modified to switch from Supervisor to User-0 or User-1 mode. In this case, the outstanding store transaction, executed in Supervisor mode, may be transferred to the bus in User mode (the bus systems do not discriminate

between User-0 and User-1 modes). Due to the blocking nature of load transactions and the fact that User mode code cannot modify the PSW, neither of these other situations can cause a problem.

Example

```
...
st.w [aX], dX  ; Store to Supervisor mode protected SFR
mtcr #PSW, dY  ; Modify PSW.IO to switch to User mode
...
```

Workaround

Any MTCR instruction targeting the PSW, which may change the PSW.IO bit field, must be preceded by a DSYNC instruction, unless it can be guaranteed that no store transaction is outstanding.

```
...
st.w [aX], dX  ; Store to Supervisor mode protected SFR
dsync
mtcr #PSW, dY  ; Modify PSW.IO to switch to User mode
...
```

CPU_TC.107 SYSCON.FCDSF may not be set after FCD Trap

Under certain conditions the SYSCON.FCDSF flag may not be set after an FCD trap is entered. This situation may occur when the CSA (Context Save Area) list is located in cacheable memory, or, dependent upon the state of the upper context shadow registers, when the CSA list is located in LDRAM.

The SYSCON.FCDSF flag may be used by other trap handlers, typically those for asynchronous traps, to determine if an FCD trap handler was in progress when the another trap was taken.

Workaround

In the case where the CSA list is statically located in memory, asynchronous trap handlers may detect that an FCD trap was in progress by comparing the

current values of FCX and LCX, thus achieving similar functionality to the SYSCON.FCDSF flag.

In the case where the CSA list is dynamically managed, no reliable workaround is possible.

CPU_TC.108 Incorrect Data Size for Circular Addressing mode instructions with wrap-around

In certain situations where a Load or Store instruction using circular addressing mode encounters the circular buffer wrap-around condition, the first access to the circular buffer may be performed using an incorrect data size, causing too many or too few data bytes to be transferred. The circular buffer wrap-around condition occurs when a load or store instruction using circular addressing mode addresses a data item which spans the boundary of a circular buffer, such that part of the data item is located at the top of the buffer, with the remainder at the base. The problem may occur in one of two cases:

Case 1

Where a **store** instruction using circular addressing mode encounters the circular buffer wrap-around condition, and is preceded in the LS pipeline by a multi-access load instruction, the first access of the store instruction using circular addressing mode may incorrectly use the transfer data size from the second part of the multi-access load instruction. A multi-access load instruction occurs in one of the following circumstances:

- Unaligned access to LDRAM or cacheable address which spans a 128-bit boundary.
- Unaligned access to a non-cacheable, non-LDRAM address.
- Circular addressing mode access which encounters the circular buffer wrap-around condition.

Since half-word store instructions must be half-word aligned, and st.a instructions must be word aligned, they cannot trigger the circular buffer wrap-around condition. As such, this case only affects the following instructions using circular addressing mode: st.w, st.d, st.da.

Example

```
...
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
st.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
...
```

In this example, the word load from address 0xD000000E is split into 2 half-word accesses, since it spans a 128-bit boundary in LDRAM. The double-word store encounters the circular buffer wrap condition and should be split into 2 word accesses, to the top and bottom of the circular buffer. However, due to the bug, the first access takes the transfer data size from the second part of the un-aligned load and only 16-bits of data are written. Note that the presence of an optional IP instruction between the load and store transactions does not prevent the problem, since the load and store transactions are back-to-back in the LS pipeline.

Case 2

Case 2 is similar to case 1, and occurs where a **load** instruction using circular addressing mode encounters the circular buffer wrap-around condition, and is preceded in the LS pipeline by a multi-access load instruction. However, for case 2 to be a problem it is necessary that the first access of the load instruction encountering the circular buffer wrap-around condition (the access to the top of the circular buffer) also encounters a conflict condition with the contents of the CPU store buffer. Again, in this case the first access of the load instruction using circular addressing mode may incorrectly use the transfer data size from the second part of the multi-access load instruction. Since half-word load instructions must be half-word aligned, and ld.a instructions must be word aligned, they cannot trigger the circular buffer wrap-around condition. As such, this case only affects the following instructions using circular addressing mode: ld.w, ld.d, ld.da.

Note: In the current TriCore1 CPU implementation, load accesses are initiated from the DEC pipeline stage whilst store accesses are initiated from the following EXE pipeline stage. To avoid memory port contention problems when a load follows a store instruction, the CPU contains a single store buffer. In the case where a store instruction (in EXE) is immediately followed by a load instruction (in DEC), the store is directed to the CPU store buffer and the load operation overtakes the store. The store is then committed to memory from the store buffer on the next store instruction or non-memory access cycle. The store buffer is only used for store accesses to 'local' memories - LDRAM or DCache. Store instructions to bus-based memories are always executed immediately (in-order). A store buffer conflict is detected when a load instruction is encountered which targets an address for which at least part of the requested data is currently held in the CPU store buffer. In this store buffer conflict scenario, the load instruction is cancelled, the store committed to memory from the store buffer and then the load re-started. In systems with an enabled MMU and where either the store buffer or load instruction targets an address undergoing PTE-based translation, the conflict detection is just performed on address bits (9:0), since higher order bits may be modified by translation and a conflict cannot be ruled out. In other systems (no MMU, MMU disabled), conflict detection is performed on the complete address.

Example

```
...
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
st.h [a12]0x14, d7   ; Store causing conflict
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
ld.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
                        ; conflict with st.h
...
```

In this example, the half-word store is to address 0xD0000834 and is immediately followed by a load instruction, so is directed to the store buffer. The

word load from address 0xD000000E is split into 2 half-word accesses, since it spans a 128-bit boundary in LDRAM. The double-word load encounters the circular buffer wrap condition and should be split into 2 word accesses, to the top and bottom of the circular buffer. In addition, the first circular buffer access conflicts with the store to address 0xD0000834. Due to the bug, after the store buffer is flushed, the first access takes the transfer data size from the second part of the un-aligned load and only 16-bits of data are read. Note that the presence of an optional IP instruction between the two load transactions does not prevent the problem, since the load transactions are back-to-back in the LS pipeline.

Workaround

Where it cannot be guaranteed that a word or double-word load or store instruction using circular addressing mode will not encounter one of the corner cases detailed above which may lead to incorrect behaviour, one NOP instruction should be inserted prior to the load or store instruction using circular addressing mode.

```
...
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
nop                  ; Bug workaround
st.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
...
```

CPU_TC.109 Circular Addressing Load can overtake conflicting Store in Store Buffer

In a specific set of circumstances, a load instruction using circular addressing mode may overtake a conflicting store held in the TriCore1 CPU store buffer. The problem occurs in the following situation:

- The CPU store buffer contains a **byte** store instruction, st.b, targeting the base address + 0x1 of a circular buffer.
- A **word** load instruction, ld.w, is executed using circular addressing mode, targetting the same circular buffer as the buffered byte store.
- This word load is only half-word aligned and encounters the circular buffer wrap-around condition such that the second, wrapped, access of the load instruction to the bottom of the circular buffer targets the same address as the byte store held in the store buffer.

Additionally, one of the following further conditions must also be present for the problem to occur:

- The circular buffer base address for the word load is double-word but not quad-word (128-bit) aligned - i.e. the base address has bits (3:0) = 0x8 with the conflicting byte store having address bits (3:0) = 0x9, OR,
- The circular buffer base address for the word load is quad-word (128-bit) aligned and the circular buffer size is an odd number of words - i.e. the base address has bits (3:0) = 0x0 with the conflicting byte store having address bits (3:0) = 0x1.

In these very specific circumstances the conflict between the load instruction and store buffer contents is not detected and the load instruction overtakes the store, returning the data value prior to the store operation.

Note: In the current TriCore1 CPU implementation, load accesses are initiated from the DEC pipeline stage whilst store accesses are initiated from the following EXE pipeline stage. To avoid memory port contention problems when a load follows a store instruction, the CPU contains a single store buffer. In the case where a store instruction (in EXE) is immediately followed by a load instruction (in DEC), the store is directed to the CPU store buffer and the load operation overtakes the store. The store is then committed to memory from the store buffer on the next store instruction or non-memory access cycle. The store buffer is only used for store accesses to 'local' memories - LDRAM or DCache. Store instructions to bus-based memories are always executed immediately (in-order). A store buffer conflict is detected when a load instruction is encountered which targets an address for which at least part of the requested data is currently held in the CPU store buffer. In this store buffer conflict scenario, the load instruction is cancelled, the store committed to memory from the store

buffer and then the load re-started. In systems with an enabled MMU and where either the store buffer or load instruction targets an address undergoing PTE-based translation, the conflict detection is just performed on address bits (9:0), since higher order bits may be modified by translation and a conflict cannot be ruled out. In other systems (no MMU, MMU disabled), conflict detection is performed on the complete address.

Example - Case 1

```
...
LDA  a12, 0xD0001008 ; Circular Buffer Base
LDA  a13, 0x00180016 ; Circular Buffer Limit and Index
...
st.b [a12]0x1, d2    ; Store to byte offset 0x9
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

In this example the circular buffer base address is double-word but not quad-word aligned. The byte store to address 0xD0001009 is immediately followed by a load operation and is placed in the CPU store buffer. The word load instruction encounters the circular buffer wrap condition and is split into 2 half-word accesses, to the top (0xD0001016) and bottom (0xD0001008) of the circular buffer. The first load access completes correctly, but, due to the bug, the second access overtakes the store operation and returns the previous half-word from 0xD0001008.

Example - Case 2

```
...
LDA  a12, 0xD0001000 ; Circular Buffer Base
LDA  a13, 0x00140012 ; Circular Buffer Limit and Index
...
st.b [a12]0x1, d2    ; Store to byte offset 0x1
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

In this example the circular buffer base address is quad-word aligned but the buffer size is an odd number of words (0x14 = 5 words). The byte store to address 0xD0001001 is immediately followed by a load operation and is placed

in the CPU store buffer. The word load instruction encounters the circular buffer wrap condition and is split into 2 half-word accesses, to the top (0xD0001012) and bottom (0xD0001000) of the circular buffer. The first load access completes correctly, but, due to the bug, the second access overtakes the store operation and returns the previous half-word from 0xD0001000.

Workaround

For any circular buffer data structure, if byte store operations (st.b) are not used targeting the circular buffer, or if the circular buffer has a quad-word aligned base address and is an even number of words in depth, then this problem cannot occur. If these restrictions and the other conditions required to trigger the problem cannot be ruled out, then any load word instruction (ld.w) targeting the buffer using circular addressing mode, and which may encounter the circular buffer wrap condition, must be preceded by a single NOP instruction.

```
...
LDA  a12, 0xD0001000 ; Circular Buffer Base
LDA  a13, 0x00140012 ; Circular Buffer Limit and Index
...
st.b [a12]0x1, d2    ; Store to byte offset 0x1
nop                  ; Workaround
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

CPU_TC.112 Unreliable result for MFCR read of Program Counter (PC)

The TriCore1 CPU contains a Program Counter (PC) Core Special Function Register (CSFR), which may be read either by a debugger or by usage of the MFCR instruction from a running program. According to the TriCore architecture manual, revision V1.3.8 and earlier, the PC holds the address of the instruction that is currently running.

For TriCore1 implementations up to and including TriCore1.3, independent of the method used to read the CSFR, the value returned for the PC is the address of the next instruction available from the Fetch pipeline stage. In the case of reading the PC from a debugger, with the TriCore1 CPU halted, then this is the address of the next instruction that will be executed once the CPU is re-started

(excluding interrupt conditions) and is always correctly supplied. However, when reading the PC from a running program using the MFCR instruction, the address of the next instruction available from the Fetch pipeline stage is not architecturally defined. Instead it is an implementation specific value dependent on the successive instructions, code alignment, cache hit/miss conditions, code branches or interrupts; and so while repeatable (excluding interrupt conditions) is not easily determinable and made use of in general.

Workaround

Where the reliable determination of the current program counter address is required by a running program, for instance where PC-relative addressing of data is required, then one of the methods described in the section **"PC-relative Addressing"** of the TriCore1 Architecture manual must be used. For instance, in the case of dynamically loaded code, the appropriate way to load a code address for use in PC-relative addressing is to use the JL (Jump and Link) instruction. A jump and link to the next instruction is executed, placing the address of that instruction into the return address (RA) register A[11]. Before this is done though, it is necessary to copy the actual return address of the current function to another register.

Note: From the TriCore1.3.1 implementation onwards, an MFCR read of the PC CSFR will always return the address of the MFCR instruction itself.

DMA_TC.004 Reset of registers OCDSR and SUSPMR is connected to FPI reset

The reset of the debug related registers OCDSR and SUSPMR should be connected to OCDS reset according to the specification. Instead of this, their reset is connected to the normal FPI reset, i.e. these registers get reset with a normal FPI reset.

Workaround

Re-initialize the (modified) OCDSR and SUSPMR register contents whenever a FPI reset has been performed.

DMA_TC.005 Do not access MExPR, MExAENR, MExARR with RMW instructions

The DMA registers MExPR, MExAENR and MExARR are showing a misbehaviour when being accessed with LDMST or ST.T instructions.

Workaround

Do not access these registers with RMW-instructions (Read/Modify/Write). Use normal write instructions instead.

DMA_TC.007 CHSRmn.LXO bit is not reset by channel reset

The software can request a channel reset with register bit CHRSTR.CHmn. In contrast to the specification the bit CHSRmn.LXO (pattern search result flag) is not reset.

Workaround

Perform a dummy move with a known non-matching pattern to clear it.

DMA_TC.010 Channel reset disturbed by pattern found event

There is a corner case where a software triggered channel reset request collides with a concurrently running pattern found event. If both operations occur at the same time, the channel will be reset as usual, but the pattern found event will cause the destination address in DADR register to be incremented/decremented once more.

Workaround

1. When using pattern matching always issue two channel reset operations.
2. The occurrence of this corner case can be detected by software (incorrect DADR value). In this case a second channel reset request is needed.

DMA_TC.011 Pattern search for unaligned data fails on certain patterns

The DMA can be programmed to search for a pattern while doing a DMA transfer. It can search also for pattern which are distributed across 2 separate DMA moves, so called unaligned pattern. In this case the DMA stores the match result of a move in the bit CHSRmn.LX0.

Example: search unaligned for byte 0x0D followed by byte 0x0A
first move found 0x0D => CHSRmn.LX0 is set to '1'
second move found 0x0A => found & LX0='1' => pattern found

Problem description:

Once LX0 is set it will be cleared with the next move, no matter if there is another match or not. This causes pattern not to be found when the first match occurs twice in the DMA data stream.

Example: search unaligned for byte 0x0D followed by byte 0x0A
first move found 0x0D => CHSRmn.LX0 is set to '1'
second move found 0x0D => LX0 cleared
third move found 0x0A => pattern NOT found !!

Workaround

Search only for the second half of the pattern. If a match occurs check by software if it is preceded by the first half of the pattern.

DMA_TC.012 No wrap around interrupt generated

If the buffer size of a DMA channel is set to its maximum value (=32kbytes, bit field ADRCRmn.CBLx = 0xF), then no address wrap around interrupts will be generated for this channel.

Workaround

None.

DMI_TC.005 DSE Trap possible with no corresponding flag set in DMI_STR

Under certain circumstances it is possible for a DSE trap to be correctly taken by the CPU but no corresponding flag is set in the DMI Synchronous Trap flag Register (DMI_STR). The problem occurs when an out-of-range access is made to the Data ScratchPad RAM (DSPR), which would ordinarily set the DMI_STR.LRESTF flag.

If an out-of-range access is made in cycle N, but cancelled, and followed by a second out-of-range access in cycle N+1, the edge detection logic associated with the DMI_STR register fails and no flag is set.

Workaround

If a DSE trap occurs with no associated flag set in the DMI_STR register, software should treat this situation as if the DMI_STR.LRESTF flag was set.

Ethernet_TC.007 Incorrect MAC behavior when handling dribble bits

When MAC receives a frame from external PHY with additional 4-bit nibble data after CRC, it should detect a frame alignment error and CRC error and set bits MAXRXSTAT.ALIGNE and MAXRXSTAT.CRCE. But the current MAC ignore this addtional nibble data when checking CRC and detects correct CRC and sets MAXRXSTAT.GOOD. It means a frame is recieved successfully. Moreover, the MAC takes this nibble data as an additional Byte and therefore writes one more Byte to Descriptor data memory.For example, a 64 byte frame is tranmitted to MAC, 79 (6 + 6 + 2 + 64 + 1) bytes are written into Descriptor data memory instead of 78 bytes due to this incorrect MAC behavior.

Workaround

This behavior can be avoided by application software since it knows the correct length of the frame..

Ethernet_TC.008 IData might be lost when descriptor contains data less than 14 bytes in multiple-descriptor frame

If one frame consists of more than one descriptor and the data in first descriptor is less than header length (14 bytes), the data in the second descriptor is lost.

Workaround

Do not describe the data which is less than 14 bytes in one descriptor

FPU_TC.001 FPU flags always update with FPU exception

SCU_STAT latches the value of the FPU flags each time there is an FPU exception. This will overwrite the information stored in the SCU_STAT, which correspond to the first exception before the user read the information.

Workaround

None.

IIC_AI.001 Handling of the Receive/Transmit Buffer RTB

Due to incorrect internal handling of the Transmit Byte Counter (bit field CO), the following problems will occur:

1. **Unexpected IRQD in slave on stop/restart condition:** If the master receiver wants to stop the transfer, it must signal to the slave the end of data by not generating an acknowledge on the last byte. The slave generates the interrupts IRQD, IRQE and releases the data line. With getting the stop condition (generated by the master) the slave generates both interrupts IRQE and IRQD for a second time, instead of IRQE only.
2. **Incorrect buffer handling by master receiver:** If the master receiver was configured with 1 byte buffer ($CI=00_B$) the first transmit byte of data from slave will not be written to its register RTB0. If more than 1 byte was selected, the first byte is written to RTB1.

Workaround for 1.

At the first IRQE interrupt (for the no acknowledge), perform as many dummy (byte or word) reads to RTB as are required to read the number of bytes indicated in bit field CO. This resets bit field CO, because these bits are not writeable (bit IRQD will also be cleared if bit INT=0, otherwise it must be cleared by software).

Workaround for 2.

Perform a dummy read to RTB after the slave address was transmitted and bit TRX was cleared (bit IRQD will be cleared if bit INT=0, otherwise it must be cleared by software).

IIC_AI.003 SCL low time tLOW violated before repeated start condition in standard mode

With setting bit RSC and clearing interrupt flags (IRQD, IRQE, IRQP) the master wants to send a repeated start condition. The SCL line is released, but the minimum SCL low hold time is not met (SCL to low is asserted already, if SDA line is high and the minimum low time is not elapsed). Actually it is only 1/4 of the period defined by the baudrate instead of 4.7 us (as defined in the IIC standard).

Workaround

If this is not tolerated, use baudrate <= 50 kbit/s for standard mode.

IIC_AI.005 Restart condition only possible with CI = 0

If the (Multi) Master generates a repeated start condition with transmit buffer length CI > 0, then erroneously further repeated start conditions are generated for each transmitted byte (RTBx).

Workaround

Set CI to 0 for repeated start condition (RSC = 1).

MLI_TC.006 Receiver address is not wrapped around in downward direction

Overview:

- An MLI receiver performs accesses to an user defined address range, which is represented as a wrap around buffer.
- "Optimized frames" are frames without address information. The built-in address prediction defines the target address which is based on the previous address delta.
- If a buffer boundary is exceeded, the address has to be wrapped around to the opposite boundary, so that the accessed space is always within the buffer.
- An MLI transmitter will stop generating optimized frames if a user performs a wrap around access sequence in a transfer window.

Problem:

Only if a non-MLI transmitter (for example, software implemented) sends an optimized frame to a MLI receiver, but crossing the buffer boundaries, the MLI receiver will:

- Wrap around if the top limit is exceeded (upward direction).
- Access an address out of the buffer if the bottom limit is exceeded (downward direction).

The second behaviour is erroneous, as a wrap around should be performed.

Note: The hardware implemented MLI transmitter in the existing Infineon devices will not use optimized frames if a user performs a wrap around access sequence in a transfer window.

Workaround

A (software implemented) non-MLI transmitter should use non-optimized frames when crossing buffer boundaries.

MLI_TC.007 Answer frames do not trigger NFR interrupt if RIER.NFRIE=10_B and Move Engine enabled

If RIER.NFRIE=10_B, a NFR interrupt is generated whenever a frame is received but, if Move Engine is enabled (RCR.MOD=1_B, "automatic mode"), the NFR interrupt is suppressed for read/write/base frames. However, this interrupt is actually also supressed for answer frames, which are not serviced by Move Engine.

Workaround

To trigger NFR interrupts for read answer frames, having Move Engine enabled, then:

- Set RIER.NFRIE=00_B when no read is pending.
- Set RIER.NFRIE=01_B when a read is pending. Any read/write/base/answer frame will trigger the NFR interrupt. Then, by reading RCR.TF in the interrupt handler, it can be detected whether the received frame was the expected answer frame or not.

MLI_TC.008 Move engines can not access address $F01E0000_H$

DMA/MLI move engines are not able to access the address $F01E0000_H$, which represents the first byte of the small transfer window of pipe 0 in MLI0 (MLI0_SP0). If a DMA/MLI move engine access to this address is performed, the move engine will be locked.

Workaround

MLI_TC.009 MLI0B and internal loopback option not available for TC1130.

It is mentioned that MLI0B and internal Loopback mode for both MLI0 and MLI1 are available for TC1130. However, the pin RxCLK[3], RVALID[3] and RDATA[3] are wired to '0' and thus loopback mode is not possible. Likewise, the MLI0_RREADY[1] and MLI0_TVALID[1] are not connected to P4_5 and P4_2

and thus MLI0B is not available.What is connected to P4_5 and P4_2 is MLI0_RREADY[0] and MLI0_TVALID[0].

Workaround

There is no workaround for the internal loopback. To utilise MLI0B, TVALIDA and RREADYA should be used instead of TVALIDB and RREADYB.

MultiCAN_AI.040 Remote frame transmit acceptance filtering error

Correct behaviour:

Assume the MultiCAN message object receives a remote frame that leads to a valid transmit request in the same message object (request of remote answer), then the MultiCAN module prepares for an immediate answer of the remote request. The answer message is arbitrated against the winner of transmit acceptance filtering (without the remote answer) with a respect to the priority class (MOARn.PRI).

Wrong behaviour:

Assume the MultiCAN message object receives a remote frame that leads to a valid transmit request in the same message object (request of remote answer), then the MultiCAN module prepares for an immediate answer of the remote request. The answer message is arbitrated against the winner of transmit acceptance filtering (without the remote answer) with a respect to the CAN arbitration rules and not taking the PRI values into account.

If the remote answer is not sent out immediately, then it is subject to further transmit acceptance filtering runs, which are performed correctly.

Workaround

Set MOFCRn.FRREN=1_B and MOFGPRn.CUR to this message object to disable the immediate remote answering.

MultiCAN_AI.041 Dealloc Last Obj

When the last message object is deallocated from a list, then a false list object error can be indicated.

Workaround

- Ignore the list object error indication that occurs after the deallocation of the last message object.

or

- Avoid deallocating the last message object of a list.

MultiCAN_AI.042 Clear MSGVAL during transmit acceptance filtering

Assume all CAN nodes are idle and no writes to MOCTRn of any other message object are performed. When bit MOCTRn.MSGVAL of a message object with valid transmit request is cleared by software, then MultiCAN may not start transmitting even if there are other message objects with valid request pending in the same list.

Workaround

- Do not clear MOCTRn.MSGVAL of any message object during CAN operation. Use bits MOCTRn.RXEN, MOCTRn.TXEN0 instead to disable/reenable reception and transmission of message objects.

or

- Take a dummy message object, that is not allocated to any CAN node. Whenever a transmit request is cleared, set MOCTRm.TXRQ of the dummy message object thereafter. This retriggers the transmit acceptance filtering process.

MultiCAN_AI.043 Dealloc Previous Obj

Assume two message objects m and n (message object n = MOCTRm.PNEXT, i.e. n is the successor of object m in the list) are allocated. If message m is

reallocated to another list or to another position while the transmit or receive acceptance filtering run is performed on the list, then message object n may not be taken into account during this acceptance filtering run. For the frame reception message object n may not receive the message because n is not taken into account for receive acceptance filtering. The message is then received by the second priority message object (in case of any other acceptance filtering match) or is lost when there is no other message object configured for this identifier.For the frame transmission message object n may not be selected for transmission, whereas the second highest priority message object is selected instead (if any). If there is no other message object in the list with valid transmit request, then no transmission is scheduled in this filtering round. If in addition the CAN bus is idle, then no further transmit acceptance filtering is issued unless another CAN node starts a transfer or one of the bits MSGVAL, TXRQ, TXEN0, TXEN1 is set in the message object control register of any message object.

Workaround

- After reallocating message object m, write the value one to one of the bits MSGVAL, TXRQ, TXEN0, TXEN1 of the message object control register of any message object in order to retrigger transmit acceptance filtering.
- For frame reception, make sure that there is another message object in the list that can receive the message targeted to n in order to avoid data loss (e.g. a message object with an acceptance mask=0_D and PRI=3_D as last object of the list).

MultiCAN_AI.044 RxFIFO Base SDT

If a receive FIFO base object is located in that part of the list, that is used for the FIFO storage container (defined by the top and bottom pointer of this base object) and bit SDT is set in the base object (CUR pointer points to the base object), then MSGVAL of the base object is cleared after storage of a received frame in the base object without taking the setting of MOFGPRn.SEL into account.

Workaround

Take the FIFO base object out of the list segment of the FIFO slave objects, when using Single Data Transfer.

MultiCAN_AI.045 MOVIE Unexpected Interrupt

When a gateway source object or a receive FIFO base object with MOFCRn.OVIE set transmits a CAN frame, then after the transmission an unexpected interrupt is generated on the interrupt line as given by MOIPRm.RXINP of the message object referenced by m=MOFGPRn.CUR.

Workaround

Do not transmit any CAN message by receive FIFO base objects or gateway source objects with bit MOFCRn.OVIE set.

MultiCAN_AI.046 Transmit FIFO base Object position

If a message object n is configured as transmit FIFO base object and is located in the list segment that is used for the FIFO storage container (defined by MOFGPRn.BOT and MOFGPRn.TOP) but not at the list position given by MOFGPRn.BOT, then the MultiCAN uses incorrect pointer values for this transmit FIFO.

Workaround

The transmit FIFO works properly when the transmit FIFO base object is either at the bottom position within the list segment of the FIFO (MOFGPRn.BOT=n) or outside of the list segment as described above.

MultiCAN_TC.024 Power-on recovery

When Bit NCR.INIT is cleared by software (cannot be cleared by hardware in MultiCAN), MultiCAN is requested to take part in CAN traffic. Before a CAN node is allowed to take part in CAN traffic, the CAN protocol requires the CAN

node to monitor 11 consecutive recessive bits. In the MultiCAN implementation a dedicated state called “POWERON” is used to cover this waiting time.

After this waiting time has completely elapsed, the MultiCAN node leaves the POWERON state and is capable of normal CAN operations (including listen mode).

The POWERON state can be reentered only by a module reset or by setting bit `NCR.INIT`.

In the POWERON state the MultiCAN node uses a counter to count the number of consecutive samples of the receive input line. The counter is reset each time a 0 (dominant level) is found at the sample point of a bit time, and it is incremented by one each time a 1 (recessive level) is found at the sample time.

While bit `NCR.INIT` is set, the counter is forced to 0 and the MultiCAN node cannot leave POWERON state.

In the POWERON state, hard synchronization of bit timing is enabled. This means that the internal bit timing is restarted with a received dominant edge. As a result, the bit timings of the CAN bus participants are synchronized.

Correct behaviour

When the MultiCAN node is in the POWERON state, it permanently sends a recessive level at its transmit output.

Erroneous behaviour

An error occurs if the following conditions are all met:

1. MultiCAN is in the POWERON state.
2. MultiCAN is requested to transmit a message (i.e. the transfer conditions in the MultiCAN specifications are fulfilled).
3. MultiCAN has monitored 10 consecutive recessive bits.
4. MultiCAN monitors a dominant value at the sample point of the eleventh bit.

(if one of these conditions is not met, then the problem does not occur).

Then MultiCAN sends a single dominant bit after it has reached the end of the eleventh bit. Condition 4 can appear if another CAN bus participant starts to send a message before MultiCAN has reached the sample point of its eleventh bit of POWERON. In this case the single dominant bit erroneously transmitted by the MultiCAN node appears during the first identifier bit of the current

transmitter. If the MSB of the identifier of the current transmitter is also dominant, then no error occurs. If, however, the MSB of the identifier is recessive, then the current transmitter loses bus arbitration and becomes receiver (transmit line becomes recessive).

As the MultiCAN node stays in the POWERON state (because it has not seen 11 consecutive recessive bits), the MultiCAN node does not act as a transmitter to complete a started frame, but drives recessive levels at its transmit line. With the 6th recessive bit following the MSB of the identifier, other CAN bus participants detect a stuff error and transmit an error frame as a consequence.

The falling edge of the error frame leads to a resynchronization of the bit timing, assuming that at least one CAN bus participant is error active.

Due to the fact that all CAN nodes detect the stuff bit error at the same bit position, the error frame has an effective length of 6 dominant bit times, followed by 8 recessive bit times of the error delimiter and another 3 recessive bit times of interframe space.

Under normal operation conditions, a transmitter can send the SOF bit of a new frame earliest after the 3rd recessive bit of interframe space. This means that the MultiCAN nodes receives the eleven consecutive bits needed to leave POWERON state. In this scenario, the POWERON state is left correctly and normal CAN bus operation can start.

If, however, the baud rates of the MultiCAN node and the transmitter node are not perfectly matched and the MultiCAN node runs slower than the transmitter node, then the MultiCAN node could again detect the SOF bit of the transmitter at the eleventh bit of its POWERON state. Error behaviour see above.

Workaround

Workaround A

The purpose of this workaround is to prevent the MultiCAN node from receiving a dominant level while it is in the POWERON state.

Assume that bit NCR.INIT is set in the MultiCAN node, i.e. the MultiCAN node is either in the POWERON state or in the BUSOFF state.

To enable CAN operation of the MultiCAN node, the following steps need to be performed:

1. If Bit NSR.BOFF = 1, then wait until NSR.BOFF = 0 (i.e. a running bus off recovery sequence is finished correctly).
2. Disconnect the MultiCAN node from the CAN bus and connect it to the internal loop back bus by means of setting bit NPCR.LBM = 1. Please note that register NPCR is write protected by bit NCR.CCE. Make sure that no other active MultiCAN node is connected to the loop back bus, i.e. bit NPCR.LBM = 0 in all other MultiCAN nodes with NCR.INIT = 0.
3. Clear bit NCR.INIT.
4. Configure a dummy message object to transmit a dummy (remote) message on the loop back bus. As no other MultiCAN node is connected to the loop back bus, a message sent on this bus will never be acknowledged and will thus lead to an acknowledge error. This acknowledge error is indicated by NSR.LEC = 011 and an alert interrupt, if enabled. The occurrence of an acknowledge error implies that the MultiCAN node is no longer in the POWERON state and the dummy message can be disabled. This method does not need a counter and is purely event based.
5. Reconnect the MultiCAN node to the CAN bus pins by means of clearing bit NPCR.LBM.

Please note that with step 5 an ongoing message on the CAN bus by another transmitting node or of the MultiCAN node (due to a valid message object for transmission) might be corrupted. This behaviour occurs only once and is self reparing because the error condition is detected on the CAN bus and the corrupted message will be sent again automatically.

Workaround B

The purpose of this workaround is to prevent clearing the INIT bit while transmit requests are pending for the node.

1. Before clearing the INIT bit, the software has to check if there are any transmit requests pending (bits TXRQ), store pending bits (in user RAM) and clear the related pending bits TXRQs in the MultiCAN module.
2. Clear INIT bit.
3. EITHER:
   a) Clear RXOK bit and wait for RXOK to be set after a correct frame on the bus. Clear RXOK again and wait for the second correct frame on the bus, OR

b) Wait until 350 bit times (more than twice the maximum length of a CAN frame) have ellapsed.

4. Restore the previously saved transmit request bits.

MultiCAN_TC.025 RXUPD behavior

When a CAN frame is stored in a message object, either directly from the CAN node or indirectly via receive FIFO or from a gateway source object, then bit MOCTR.RXUPD is set in the message object before the storage process and is automatically cleared after the storage process.

Problem description

When a standard message object (MOFCR.MMC) receives a CAN frame from a CAN node, then it processes its own RXUPD as described above (correct).

In addition to that, it also sets and clears bit RXUPD in the message object referenced by pointer MOFGPR.CUR (wrong behavior).

Workaround

The “foreign” RXUPD pulse can be avoided by initializing MOFGPR.CUR with the message number of the object itself instead of another object (which would be message object 0 by default, because MOFGPR.CUR points to message object 0 after reset initialization of MultiCAN).

MultiCAN_TC.026 MultiCAN Timestamp Function

The timestamp functionality does not work correctly.

Workaround

Do not use timestamp.

MultiCAN_TC.027 MultiCAN Tx Filter Data Remote

Message objects of priority class 2 (MOAR.PRI = 2) are transmitted in the order as given by the CAN arbitration rules. This implies that for 2 message objects which have the same CAN identifier, but different DIR bit, the one with DIR = 1 (send data frame) shall be transmitted before the message object with DIR = 0, which sends a remote frame. The transmit filtering logic of the MultiCAN leads to a reverse order, i.e the remote frame is transmitted first. Message objects with different identifiers are handled correctly.

Workaround

None.

MultiCAN_TC.028 SDT behavior

Correct behavior

Standard message objects:

MultiCAN clears bit MOCTR.MSGVAL after the successful reception/transmission of a CAN frame if bit MOFCR.SDT is set.

Transmit Fifo slave object:

MultiCAN clears bit MOCTR.MSGVAL after the successful reception/transmission of a CAN frame if bit MOFCR.SDT is set. After a transmission, MultiCAN also looks at the respective transmit FIFO base object and clears bit MSGVAL in the base object if bit SDT is set in the base object and pointer MOFGPR.CUR points to MOFGPR.SEL (after the pointer update).

Gateway Destination/Fifo slave object:

MultiCAN clears bit MOCTR.MSGVAL after the storage of a CAN frame into the object (gateway/FIFO action) or after the successful transmission of a CAN frame if bit MOFCR.SDT is set. After a reception, MultiCAN also looks at the respective FIFO base/Gateway source object and clears bit MSGVAL in the base object if bit SDT is set in the base object and pointer MOFGPR.CUR points to MOFGPR.SEL (after the pointer update).

Problem description

Standard message objects:

After the successful transmission/reception of a CAN frame, MultiCAN also looks at message object given by MOFGPR.CUR. If bit SDT is set in the referenced message object, then bit MSGVAL is cleared in the message object CUR is pointing to.

Transmit FIFO slave object:

Same wrong behaviour as for standard message object. As for transmit FIFO slave objects CUR always points to the base object, the whole transmit FIFO is set invalid after the transmission of the first element instead after the base object CUR pointer has reached the predefined SEL limit value.

Gateway Destination/Fifo slave object:

Correct operation of the SDT feature.

Workaround

Standard message object:

Set pointer MOFGPR.CUR to the message number of the object itself.

Transmit FIFO:

Do not set bit MOFCR.SDT in the transmit FIFO base object. Then SDT works correctly with the slaves, but the FIFO deactivation feature by CUR reaching a predefined limit SEL is lost.

MultiCAN_TC.029 Tx FIFO overflow interrupt not generated

Specified behaviour

After the successful transmission of a Tx FIFO element, a Tx overflow interrupt is generated if the FIFO base object fulfils these conditions:

- Bit MOFCR.OVIE=1, AND
- MOFGPR.CUR becomes equal to MOFGPR.SEL

Real behaviour

A Tx FIFO overflow interrupt will not be generated after the transmission of the Tx FIFO base object.

Workaround

If Tx FIFO overflow interrupt needed, take the FIFO base object out of the circular list of the Tx message objects. That is to say, just use the FIFO base object for FIFO control, but not to store a Tx message.

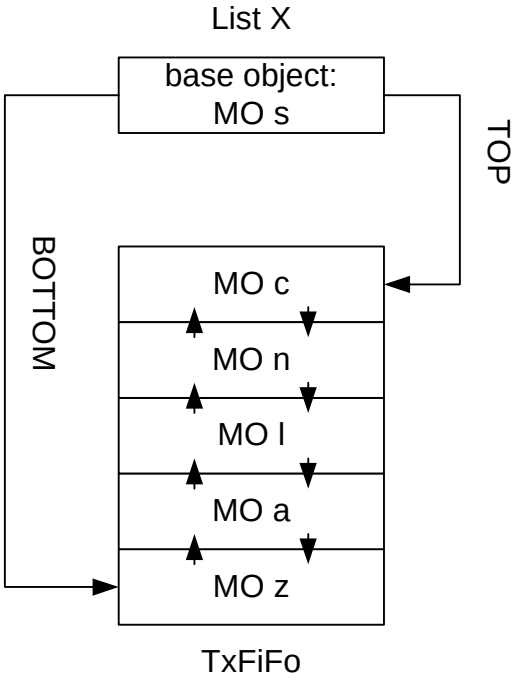

Figure 1 FIFO structure

MultiCAN_TC.030 Wrong transmit order when CAN error at start of CRC transmission

The priority order defined by acceptance filtering, specified in the message objects, define the sequential order in which these messages are sent on the CAN bus. If an error occurs on the CAN bus, the transmissions are delayed due to the destruction of the message on the bus, but the transmission order is kept. However, if a CAN error occurs when starting to transmit the CRC field, the arbitration order for the corresponding CAN node is disturbed, because the faulty message is not retransmitted directly, but after the next transmission of the CAN node.

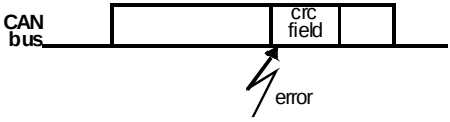

Figure 2

Workaround

None.

MultiCAN_TC.031 List Object Error wrongly triggered

If the first list object in a list belonging to an active CAN node is deallocated from that list position during transmit/receive acceptance filtering (happening during message transfer on the bus), then a "list object" error may occur ($NSRx.LOE=1_B$), which will cause that effectively no acceptance filtering is performed for this message by the affected CAN node.

As a result:

- for the affected CAN node, the CAN message during which the error occurs will not be stored in a message object. This means that although the message is acknowledged on the CAN bus, its content will be ignored.

- the message handling of an ongoing transmission is not disturbed, but the transmission of the subsequent message will be delayed, because transmit acceptance filtering has to be started again.
- message objects with pending transmit request might not be transmitted at all due to failed transmit acceptance filtering.

Workaround

EITHER:

Avoid deallocation of the first element on active CAN nodes. Dynamic reallocations on message objects behind the first element are allowed.

OR:

Avoid list operations on a running node. Only perform list operations, if CAN node is not in use (e.g. when NCRx.INIT=1_B)

MultiCAN_TC.032 MSGVAL wrongly cleared in SDT mode

When Single Data Transfer Mode is enabled (MOFCRn.SDT=1_B), the bit MOCTRn.MSGVAL is cleared after the reception of a CAN frame, no matter if it is a data frame or a remote frame.

In case of a remote frame reception and with MOFCR.FRREN = 0_B, the answer to the remote frame (data frame) is transmitted despite clearing of MOCTRn.MSGVAL (incorrect behaviour). If, however, the answer (data frame) does not win transmit acceptance filtering or fails on the CAN bus, then no further transmission attempt is made due to cleared MSGVAL (correct behaviour).

Workaround

- To avoid a single trial of a remote answer in this case, set MOFCR.FRREN = 1_B and MOFGPR.CUR = this object.

MultiCAN_TC.035 Different bit timing modes

Bit timing modes (NFCRx.CFMOD=10_B) do not conform to the specification.

When the modes 001_B-100_B are set in register NFCRx.CFSEL, the actual configured mode and behaviour is different than expected.

Table 10

Bit timing mode (NFCR.CFSEL) according to spec	Value to be written to NFCR.CFSEL instead	Measurement
001_B	Mode is missing (not implemented) in MultiCAN	Whenever a recessive edge (transition from 0 to 1) is monitored on the receive input the time (measured in clock cycles) between this edge and the most recent dominant edge is stored in CFC.
010_B	011_B	Whenever a dominant edge is received as a result of a transmitted dominant edge the time (clock cycles) between both edges is stored in CFC.
011_B	100_B	Whenever a recessive edge is received as a result of a transmitted recessive edge the time (clock cycles) between both edges is stored in CFC.
100_B	001_B	Whenever a dominant edge that qualifies for synchronization is monitored on the receive input the time (measured in clock cycles) between this edge and the most recent sample point is stored in CFC.

Workaround

None.

MultiCAN_TC.037 Clear MSGVAL

Correct behaviour:

When MSGVAL is cleared for a message object in any list, then this should not affect the other message objects in any way.

Message reception (wrong behaviour):

Assume that a received CAN message is about to be stored in a message object A, which can be a standard message object, FIFO base, FIFO slave, gateway source or gateway destination object.

If during of the storage action the user clears MOCTR.MSGVAL of message object B in any list, then the MultiCAN module may wrongly interpret this temporarily also as a clearing of MSGVAL of message object A. The result of this is that the message is not stored in message object A and is lost. Also no status update is performed on message object A (setting of NEWDAT, MSGLST, RXPND) and no message object receive interrupt is generated. Clearing of MOCTR.MSGVAL of message object B is performed correctly.

Message transmission (wrong behaviour):

Assume that MultiCAN is about to copy the message content of a message object A into the internal transmit buffer of the CAN node for transmission.

If during of the copy action the user clears MOCTR.MSGVAL of message object B in any list, then the MultiCAN module may wrongly interpret this also as a clearing of MSGVAL of message object A. The result of this is that the copy action for message A is not performed, bit NEWDAT is not cleared and no transmission takes place (clearing MOCTR.MSGVAL of message object B is performed correctly). In case of idle CAN bus and the user does not actively set the transmit request of any message object, this may lead to not transmitting any further message object, even if they have a valid transmit request set.

Single data transfer feature:

When the MultiCAN module clears MSGVAL as a result of a single data transfer (MOFCR.SDT = 1 in the message object), then the problem does not occur. The problem only occurs if MSGVAL of a message object is cleared via CPU.

Workaround

Do not clear MOCTR.MSGVAL of any message object during CAN operation. Use bits MOCTR.RXEN, MOCTR.TXEN0 instead to disable/reenable reception and transmission of message objects.

MultiCAN_TC.038 Cancel TXRQ

When the transmit request of a message object that has won transmit acceptance filtering is cancelled (by clearing MSGVAL, TXRQ, TXEN0 or TXEN1), the CAN bus is idle and no writes to MOCTR of any message object are performed, then MultiCAN does not start the transmission even if there are message objects with valid transmit request pending.

Workaround

To avoid that the CAN node ignores the transmission:

- take a dummy message object, that is not allocated to any CAN node. Whenever a transmit request is cleared, set TXRQ of the dummy message object thereafter. This retriggers the transmit acceptance filtering process.

or:

- whenever a transmit request is cleared, set one of the bits TXRQ, TXEN0 or TXEN1, which is already set, again in the message object for which the transmit request is cleared or in any other message object. This retriggers the transmit acceptance filtering process.

OCDS_TC.007 DBGSR writes fail when coincident with a debug event

When a CSFR write to the DBGSR occurs in the same cycle as a debug event, the write data is lost and the DBGSR updates from the debug event alone.CSFR writes can occur as the result of a MTCR instruction or an FPI write transaction from an FPI master such as Cerberus.

Workaround

Writes to the DBGSR cannot be guaranteed to occur. Following a DBGSR write the DBGSR should be read to ensure that the write was successful, and take an appropriate action if it was not. The action of the simultaneous debug event will have to be considered when determining whether to repeat the DBGSR write, do nothing, or perform some other sequence.

Writes to the DBGSR are almost always to put the TriCore either into, or out of, halt mode. Since the TriCore can not release itself from halt mode, and only rarely puts itself into halt mode, DBGSR writes are usually made by Cerberus.

Example 1 The processor executes a MFCR instruction when a DBGSR write from Cerberus occurs that attempts to put the core into halt mode. The core register debug event occurs and CREVT.EVTA = 001B so the breakout signal is pulsed. The write from Cerberus is unsuccessful and TriCore continues executing. Implementing the workaround, Cerberus reads the DBGSR to check that halt mode has been entered. Since this time it has not, the DBGSR write is repeated as is the read. If the read now indicates that the second DBGSR write was successful and TriCore is now in halt mode, the process driving Cerberus may continue.

Example 2 The processor executes a DEBUG instruction when a DBGSR write from Cerberus occurs that attempts to put the core into halt mode. The software debug event occurs and SWEVT.EVTA = 010B so TriCore enters halt mode and the breakout signal is pulsed. The write from Cerberus did not occur, but the TriCore does enter halt mode. Cerberus reads DBGSR and continues since the TriCore is now halted.

Example 3 The processor is halted, an external debug event occurs when a DBGSR write from Cerberus occurs that attempts to release the core from halt mode. The external debug event occurs and EXEVT.EVTA = 001B so the breakout signal is pulsed. The write from Cerberus does not occur and TriCore remains in halt mode. Cerberus reads DBGSR to determine if its write was successful, it was not, so it repeats the write. This time the write was successful, and TriCore is released from halt. Cerberus reads the DBGSR to confirm that the second write succeeded and moves on.

OCDS_TC.008 Breakpoint interrupt posting fails for ICR modifying instructions

BAM debug events with breakpoint interrupt actions which occur on instructions which modify ICR.CCPN or ICR.IE can fail to correctly post the interrupt. The breakpoint interrupt is either taken or posted based on the ICR contents before the instruction before the instruction rather than after the instruction, as required for a BAM debug event. The breakpoint interrupt may be posted when it should be taken or vice versa.

BAM breakpoint interrupts occurring on an MTCR, SYSCALL, RET, RFE, RSLCX, LDLCX and LDUCX instructions may be affected.

Workaround

None.

OCDS_TC.009 Data access trigger events unreliable

Trigger events set on data accesses do not fire reliably. Whilst they may sometimes successfully generate trigger events, they often will not.

Workaround

None.

Debug triggers should only be used to create trigger events on instruction execution.

OCDS_TC.010 DBGSR.HALT[0] fails for separate resets

When TriCore's main reset and debug reset are not asserted together DBGSR.HALT[0] can fail to indicate whether the CPU is in halt mode or not. This is because the halt mode can be entered or exited when a main reset occurs, depending on the boot halt signal. However DBGSR is reset when debug reset is asserted.

Example 1 TriCore is in halt mode and DBGSR.HALT[0] = '1'. The main reset signal is asserted, and boot halt is negated, so TriCore is released from halt

mode. However, because debug reset was not asserted DBGSR.HALT[0] = ’1’ incorrectly.

Example 2 TriCore is executing code (not in halt mode) and DBGSR.HALT[0] = ’0’. The main reset signal is asserted, and boot halt is asserted, so TriCore enters halt mode. However, because debug reset was not asserted DBGSR.HALT[0] = ’0’ incorrectly.

Example 3 TriCore is in halt mode and DBGSR.HALT[0] = ’1’. The debug reset signal is asserted, whilst the main reset is not. TriCore remains in halt mode, however, DBGSR.HALT[0] = ’0’ incorrectly.

Workaround

None.

OCDS_TC.011 Context lost for multiple breakpoint traps

Context lost for multiple breakpoint trapsOn taking a debug trap TriCore saves a fast context (PCX,PSW,A10,A11) at the location defined by the DCX register. The DCX location is only able to store a single fast context.

When a debug event has occurred which causes a breakpoint trap to occur TriCore executes the monitor code. If another debug event with a breakpoint trap action occurs,a new fast context will be written to the location defined in the DCX and the original fast context will be lost.

Workaround

There are two parts of this workaround. Both parts must be adhered to.

1. External debug events must not be setup to have breakpoint trap actions.

2. Do not allow non-external (trigger, software and core register) debug events with breakpoint trap actions to occur within monitor code. So trigger events, software debug events, with breakpoint trap actions should not be set on the monitor code. So long as the debug events have non breakpoint actions they may be set to occur in the monitor code.

OCDS_TC.012 Multiple debug events on one instruction can be unpredictable

When more than one debug event is set to occur on a single instruction, the debug event priorities should determine which debug event is actually generated. However these priorities have not been implemented consistently.

Note: This only affects events from the trigger event unit and events from DEBUG, MTCR and MFCR instructions. The behaviour of the external debug event is not modified by this erratum.

Workaround

Trigger events must not be set to occur on DEBUG, MTCR and MFCR instructions, or on instructions which already have a trigger event set on them.

PMI_TC.001 Deadlock possible during Instruction Cache Invalidation

Deadlock of the TriCore1 processor is possible under certain circumstances when an instruction cache invalidation operation is performed. Instruction cache invalidation is performed by setting the `PMI_CON1.CCINV` special function register bit, then clearing this bit via software. Whilst `PMI_CON1.CCINV` is active the instruction Tag memories are cleared and new instruction fetches from the LMB are inhibited. Dependent upon the state of the instruction fetch bus master state machine this may lead to system deadlock, since it may not be possible to fetch the instruction to clear the `PMI_CON1.CCINV` bit if this sequence is executed from LMB based memory.

Workaround

The set and clear of the `PMI_CON1.CCINV` bit must be performed by code executing from program scratchpad memory.

SSC_AI.023 Clock phase control causes failing data transmission in slave mode

If SSC_CON.PH = 1 and no leading delay is issued by the master, the data output of the slave will be corrupted. The reason is that the chip select of the master enables the data output of the slave. As long as the chip is inactive the slave data output is also inactive.

Workaround

A leading delay should be used by the master.

A second possibility would be to initialize the first bit to be sent to the same value as the content of PISEL.STIP.

SSC_AI.024 SLSO output gets stuck if a reconfig from slave to master mode happens

The slave select output SLSO gets stuck if the SSC will be re-configured from slave to master mode. The SLSO will not be deactivated and therefore not correct for the 1st transmission in master mode. After this 1st transmission the chip select will be deactivated and working correctly for the following transmissions.

Workaround

Ignore the 1st data transmission of the SSC when changed from slave to master mode.

SSC_AI.025 First shift clock period will be one PLL clock too short because not syncronized to baudrate

The first shift clock signal duration of the master is one PLL clock cycle shorter than it should be after a new transmit request happens at the end of the previous transmission. In this case the previous transmission had a trailing delay and an inactive delay.

Workaround

Use at least one leading delay in order to avoid this problem.

SSC_AI.026 Master with highest baud rate set generates erroneous phase error

If the SSC is in master mode, the highest baud rate is initialized and CON.PO = 1 and CON.PH = 0 there will be a phase error on the MRST line already on the shift edge and not on the latching edge of the shift clock.

- Phase error already at shift edge
  The master runs with baud rate zero. The internal clock is derived from the rising and the falling edge. If the baud rate is different from zero there is a gap between these pulses of these internal generated clocks.
  However, if the baud rate is zero there is no gap which causes that the edge detection is to slow for the "fast" changing input signal. This means that the input data is already in the first delay stage of the phase detection when the delayed shift clock reaches the condition for a phase error check. Therefore the phase error signal appears.
- Phase error pulse at the end of transmission
  The reason for this is the combination of point 1 and the fact that the end of the transmission is reached. Thus the bit counter SSCBC reaches zero and the phase error detection will be switched off.

Workaround

Don’t use a phase error in master mode if the baud rate register is programmed to zero (SSCBR = 0) which means that only the fractional divider is used.
Or program the baud rate register to a value different from zero (SSCBR > 0) when the phase error should be used in master mode.

SSC_TC.008 SSC shift register not updated in fractional divider mode

Transmitted data might be corrupted, if the SSC is used together with the fractional divider mode and a former transmission is not yet finished while new

transmission data is written into the buffered transmit register. Data corruption only may occur, if write access with new data to the transmit buffer is performed in the last bit time slice, which is shifting out the last data bit at the end of the previous transmission..

Workaround

1.) Do not use the fractional divider2.) Wait for the receive interrupt instead of the transmit interruptfor sending the next data

SSC_TC.011 Unexpected phase error

If SSCCON.PH = 1 (Shift data is latched on the first shift clock edge) the data input of master should change on the second shift clock edge only. Since the slave select signals change always on the 1st edge and they can trigger a change of the data output on the slave side, a data change is possible on the 1st clock edge.

As a result of this configuration the master would activate the slave at the same time as it latches the expected data. Therefore the first data latched is might be wrong.

To avoid latching of corrupt data, the usage of leading delay is recommended. But even so a dummy phase error can be generated during leading, trailing and inactive delay, since the check for a phase error is done with the internal shift clock, which is running during leading and trailing delay even if not visible outside the module.

If external circuitry (pull devices) delay a data change in slave_out/master_in after deactivation of the slave select line for n*(shift_clock_perid/2) then a dummy phase error can also be generated during inactive delay, even if SSCCON.PH = 0.

Workaround

Don't evaluate phase error flag SSCSTAT.PE. This is no restriction for standard applications (the flag is implemented for test purpose).

SSC_TC.017 Slaveselect (SLSO) delays may be ignored

In master mode, if a transmission is started during the period between the receive interrupt is detected and the STAT.BSY bit becomes disabled (that is to say, the period while the former communication has not yet been completed), all delays (leading, trailing and inactive) may be ignored for the next transmission.

Workaround

Wait for the STAT.BSY bit to become disabled before starting next transmission. There are two ways:

3 Deviations from Electrical- and Timing Specification

4 Application Hints

IIC_AI.H003 IIC master cannot lose arbitration at acknowledge bit during read

Two Multi-Master are configured in received mode. The master sends a non acknowledge and arbitration is still active. If the remote master acknowledges the same byte, the master will not recognize the arbitration lost situation.

The remote master and the master must always request the same number of bytes from the same slave.

IIC_AI.H005 General call feature does not work with 10-bit address mode

The general call feature does not work correctly, if the slave is configured in 10-bit address mode (bit M10 = 1). The general call is either missed or the first data byte is interpreted as address byte.

IIC_AI.H006 TRX- Bit is not immediately set after writing to register RTB

Bit TRX should be set automatically after writing to the transmit buffer (RTB). If bit BUM is set immediately after RTB is written, $0xFF_H$ is written to the bus.

It is recommended to perform a dummy read of the IIC Control Register (which includes bit BUM) after writing to RTB and before setting bit BUM.

INT_TC.H001 Multiple SRNs can be assigned to the same SRPN (priority)

Some customers may want to stay with the 3 cycle arbitration they use at the moment, but more than 63 different interrupt nodes are needed. In this case, multiple SRNs can be assigned to the same SRPN (priority). As the hardware can only arbitrates the highest priority and its clear that not multiple SRNs can win. But most peripherals have interrupt flags to show which interrupt occurs

inside the status registers. These flags can be used for the software arbitration.So there are two options:Either it doesn't care which SRN wins inside a group with the same priorities.Or such groups are built only out of SRNs from peripherals, which have interrupt flags and perform some kind of software arbitration..

MultiCAN_AI.H005 TxD Pulse upon short disable request

If a CAN disable request is set and then canceled in a very short time (one bit time or less) then a dominant transmit pulse may be generated by MultiCAN module, even if the CAN bus is in the idle state.

Example for setup of the CAN disable request:

`CAN_CLC.DISR` = 1 and then `CAN_CLC.DISR` = 0

Workaround

Set all INIT bits to 1 before requesting module disable.

MultiCAN_TC.H002 Double Synchronization of receive input

The MultiCAN module has a double synchronization stage on the CAN receive inputs. This double synchronization, delays the receive data by 2 module clock cycles. If the MultiCAN is operating at a low module clock frequencies and high CAN baudrate, this delay may become significant and has to be taken into account when calculating the overall physical delay on the CAN bus (transceiver delay...).

MultiCAN_TC.H003 Message may be discarded before transmission in STT mode

If `MOFCRn.STT=1` (Single Transmit Trial enabled), bit TXRQ is cleared (TXRQ=0) as soon as the message object has been selected for transmission and, in case of error, no retransmission takes places.

Therefore, if the error occurs between the selection for transmission and the real start of frame transmission, the message is actually never sent.

Workaround

In case the transmission shall be guaranteed, it is not suitable to use the STT mode. In this case, MOFCRn.STT shall be 0.

MultiCAN_TC.H004 Double remote request

Assume the following scenario: A first remote frame (dedicated to a message object) has been received. It performs a transmit setup (TXRQ is set) with clearing NEWDAT. MultiCAN starts to send the receiver message object (data frame), but loses arbitration against a second remote request received by the same message object as the first one (NEWDAT will be set).

When the appropriate message object (data frame) triggered by the first remote frame wins the arbitration, it will be sent out and NEWDAT is not reset. This leads to an additional data frame, that will be sent by this message object (clearing NEWDAT).

There will, however, not be more data frames than there are corresponding remote requests.

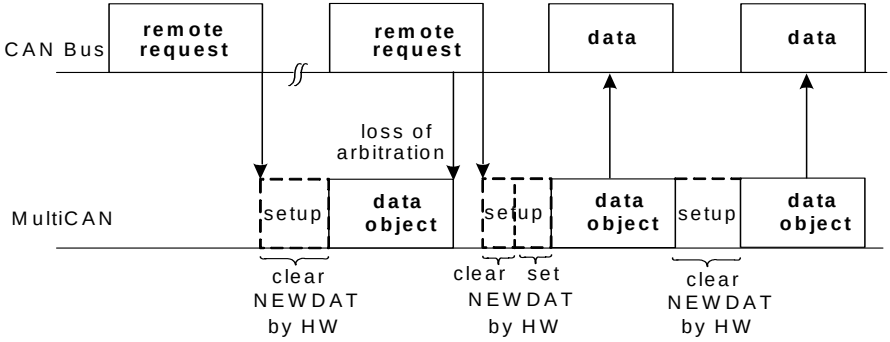

Figure 3 Loss of Arbitration

SSC_AI.H002 Transmit Buffer Update in Master Mode during Trailing or Inactive Delay Phase

When the Transmit Buffer register TB is written in master mode after a previous transmission has been completed, the start of the next transmission (generation of SCLK pulses) may be delayed in the worst case by up to 6 SCLK cycles (bit times) under the following conditions:

- a trailing delay (SSOTC.TRAIL) > 0 and/or an inactive delay (SSOTC.INACT) > 0 is configured
- the Transmit Buffer is written in the last module clock cycle (f_{SSC} or f_{CLC}) of the inactive delay phase (if INACT > 0), or of the trailing delay phase (if INACT = 0).

No extended leading delay will occur when both TRAIL = 0 and INACT = 0.

This behaviour has no functional impact on data transmission, neither on master nor slave side, only the data throughput (determined by the master) may be slightly reduced.

To avoid the extended leading delay, it is recommended to update the Transmit Buffer after the transmit interrupt has been generated (i.e. after the first SCLK phase), and before the end of the trailing or inactive delay, respectively.

Alternatively, bit BSY may be polled, and the Transmit Buffer may be written after a waiting time corresponding to 1 SCLK cycle after BSY has returned to 0_B.

After reset, the Transmit Buffer may be written at any time.

SSC_AI.H003 Transmit Buffer Update in Slave Mode during Transmission

After reset, data written to the Transmit Buffer register TB are directly copied into the shift register. Further data written to TB are stored in the Transmit Buffer while the shift register is not yet empty, i.e. transmission has not yet started or is in progress.

If the Transmit Buffer is written in slave mode during the first phase of the shift clock SCLK supplied by the master, the contents of the shift register are overwritten with the data written to TB, and the first bit currently transmitted on line MRST may be corrupted. No Transmit Error is detected in this case.

It is therefore recommended to update the Transmit Buffer in slave mode after the transmit interrupt (TIR) has been generated (i.e. after the first SCLK phase).

After reset, the Transmit Buffer may be written at any time.

SSC_TC.H002 Enlarged leading delay in master mode

f leading delay > 0 is selected in master mode, the SSC module generates slightly enlarged leading delay (< 1 shift clock cycle additional time) for a new word transfer if its TB is loaded with new data just when the former transfer ends..