Excalibur EPXA4 Devices

November 2002, ver. 1.2 Errata Sheet

EXCALIBUR™

This errata sheet provides updated information about the Excalibur™ EPXA4, revision A (see Figure 1) Devices.

Figure 1. Identify Revision A Devices

The errata fall into two categories:

- Known errata for the EPXA4 device—detailed in this document
- Known errata for the ARM922T processor provided by ARM Ltd.—detailed in Appendix A of this document

The following sections of the device are covered by errata information:

- Expansion bus interface (EBI)
- Dual-port SRAM (DPRAM)
- AHB bridges
- UART
- SDRAM
- Embedded trace module version 2a
- Configuration
- Debug module

Contact Altera® for the latest information.

EBI

This section provides further information about errata in the EBI.

1.1 Locked 16-Beat Incrementing Bursts

A locked INCR16 transfer can cause the EBI to read from a peripheral twice. This can cause erroneous behavior if the peripheral contains read-sensitive registers.

Work Around

Do not use burst transactions to access read-sensitive peripherals connected to the EBI.

1.2 EBI Acknowledge Signal

The EBI acknowledge signal, `EBI_ACK`, functionality results in incorrect EBI asynchronous mode operation.

Work Around

Do not use the `EBI_ACK` signal when interfacing to external memory devices. Instead, use the programmable wait states, via the `EBI_BLOCKn` register, for the individual EBI blocks. Also ensure that the `SA` bit for each EBI block is cleared. This effectively puts the EBI in synchronous mode. EBI synchronous mode can be used to interface with synchronous or asynchronous memories provided that the appropriate number of wait states is implemented.

DPRAM

This section provides further information about errata in the DPRAM.

2.1 Port A Clocks Are Not Pre-Balanced

If the DPRAM is configured in either a deep (×8) or a wide (×32) mode, the `dp0_2_portaclk` and `dp1_3_portaclk` clock signals are not pre-balanced. The Quartus® II fitter is unable to balance locally routed clocks and generates an error.

Work Around

Promote `dp0_2_portaclk` or the `dp1_3_portaclk` signal to a global signal and recompile the design.

2.2 Locking Mechanism is Non-Functional

Using the DPRAM locking mechanism for simultaneous accesses from the stripe and PLD results in incorrect memory accesses. Simultaneous accesses to different addresses are not affected.

Work Around

Do not use the DPRAM locking feature. Instead, use a handshake mechanism such as a semaphore to control PLD and stripe accesses to common DPRAM addresses.

AHB Bridges

This section provides further information about errata in the AHB bridges.

3.1 Back-to-back Transactions through the PLD-to-Stripe Bridge Can Cause Lock-up

If the non-sequential address phase of a new transaction occurs in the same clock cycle as the final data phase of the previous transaction, the slave interface of the PLD-to-stripe bridge issues erroneous wait states causing masters to the PLD-to-stripe bridge in the PLD to lock up. Incorrect transactions can occur on AHB2.

Work Around

Insert an IDLE address phase between all transactions.

3.2 Corrupted State of the Stripe-to-PLD Bridge after PLD Reconfiguration under Processor Control

If the `MASTER_HCLK` signal is active during PLD reconfiguration under processor control, the stripe-to-PLD bridge can become corrupted before the device enters user mode. The bridge cannot respond to AHB2 transactions, which causes the AHB2 bus to lock up. Gating the source of `MASTER_HCLK` with `INIT_DONE` does not prevent the potential lock up.

Work Arounds

To avoid corrupting the bridge during PLD reconfiguration, the clock driving `MASTER_HCLK` must be inactive before the PLD enters user mode. This can be achieved in either of the following ways:

- *Software work around*—compile the design using Quartus® II version 2.1 or higher. These versions of Quartus II route the stripe-to-PLD bridge signals specifically to avoid the bridge lock up. The routing

utilizes three extra logic elements in the device and adds minimal corresponding delay to the bridge timing. If logic element usage or bridge timing is critical to your design, you can disable the automatic Quartus II routing option by adding the following parameter to the `defparam` section of the stripe instantiation file generated by the MegaWizard®:

```
lpm_instance.xa_configuration_fix = "FALSE"
```

If this routing option is disabled, you must use the hardware work around described below to ensure correct functionality.

If you use this recommended software workaround, ensure that the **Remove Redundant Logic Cells** option is not selected for your project.

If the **Perform WYSIWYG Primitive Resynthesis** option is selected for your project, you may receive warnings that the stripe signals were not routed correctly. To eliminate the warnings, re-run the MegaWizard in Quartus II version 2.2. This creates an additional settings file (**alt_exc_stripe.esf**) to ensure that the required logic elements are implemented.

- *Hardware work around*—ensure that the external clock that drives `MASTER_HCLK` is inactive before the PLD is put into reconfiguration mode, and is enabled again only after the PLD enters user mode.

  `INIT_DONE` can be used only to activate the external clock. It should not be used to disable the clock, for the following reason: when reconfiguration begins, `INIT_DONE` does not drop to low in time to prevent the error. You must use another method to disable the external clock, such as using an external device to control the clock input.

  If a PLL in the PLD portion of the device is used to drive `MASTER_HCLK`, you must drive the PLL `CLKLK_ENA` signal with `INIT_DONE` so that the PLL output does not drive `MASTER_HCLK` before the device is in user mode. See *AN 115: Using the ClockLock & ClockBoost PLL Features in APEX Devices* for information on enabling the PLD PLLs with `INIT_DONE`.

UART

This section provides further information about errata in the UART.

4.1 UART State Undefined Following Reset

When the device comes out of reset, the UART may sometimes generate a modem interrupt and behave as though a character has been received.

Work Around

To avoid generating a modem interrupt, clear all interrupts and flush the receive and transmit FIFO buffers before interrupts from the UART are enabled.

SDRAM

This section provides further information about errata in the SDRAM.

5.1 32-bit DDR SDRAM Memories are Not Supported

The SDRAM controller in the ARM-based family (including the EPXA4) does not support interfacing with 32-bit DDR SDRAM devices that use the A8 address line for the command sequence. Also 32-bit DDR memories have one `DQS` pin; ARM-based devices require four (one per byte). Memories that use the A10 address line for the command sequence and have one `DQS` pin per byte are supported.

5.2 Improper DDR SDRAM Data Accesses for Certain Clock Ratios

Incorrect data may result from DDR SDRAM accesses if the AHB1 or AHB2 master clock is greater than 4 times faster than the SDRAM clock, SD_CLK.

Work Around

Ensure that the AHB1/2 clock frequency is less than or equal to 4 times the SDRAM clock frequency.

Embedded Trace Module Version 2a

EPXA4 devices include the ARM ETM9 version 2a. Please see the *ETM9_Rev_2a_Errata.doc* for errata on this version of the ETM9. This document is available on the ARM Limited website.

Version 2a has a different configuration code register value compared to version 1, which is used in the EPXA10 devices. The ARM Trace Tools version 1.1 does not support ETM9 version 2a. Support will be included in a new version of the trace tools planned for Q2 2002.

Configuration

This section provides further information about errata in configuration.

7.1 JTAG Configuration Error

JTAG configuration of the EPXA4 does not complete successfully if there are programmed EPC configuration devices in the JTAG chain and the EPXA4 is in boot-from-serial mode. The Quartus II Programmer reports an error stating that the CONF_DONE signal did not go high.

Work Around

Erase the EPC devices that are in the chain. This allows successful configuration in boot-from-serial-mode. Or assert the Boot_Flash pin, which sets the configuration mode to boot-from-flash. In this mode, you can successfully configure the device via JTAG.

Debug Module

This section provides further information about errata in the debug module.

8.1 DEBUG_IE_BRKPT and DEBUG_DE_WATCHPT Cause Incorrect Triggers

The DEBUG_IE_BRKPT and DEBUG_DE_WATCHPT lines do not function. If asserted, they cause breakpoints or watchpoints to trigger on the wrong instruction.

Work Around

Do not connect the DEBUG_IE_BRKPT and DEBUG_DE_WATCHPT lines in your design. The Quartus II software ties them low as appropriate.

Appendix A

Errata from ARM Ltd.

This appendix reproduces information supplied by ARM Ltd. on known errata in the ARM922T™ processor and support features. The errata have little or no impact on the use of the Excalibur device, but could be useful, on rare occasions, in understanding its interaction with third-party debugging tools. Many errata detail the work arounds which tool vendors incorporate into their products, while others inform of changes in the specification and so are included for completeness.

The information contained herein is the property of ARM Ltd. and is supplied without liability for errors or omissions. No part may be reproduced or used except as authorised by contract or other written permission. The copyright and the foregoing restriction on reproduction and use extend to all media in which this information may be embodied.

Table 1 describes the errata categories defined by ARM Ltd. and used throughout this appendix, except for A2.1 "Invalid data trace following FIFO overflow—Category 1" on page 9, which refers to a previous definition of category 1 defined by ARM Ltd.

Table 1. Errata Categories

Category	Description
1	Features which are impossible to work around and severely restrict the use of the device in all or the majority of applications rendering the device unusable.
2	Features which contravene the specified behaviour and may limit or severely impair the intended use of specified features but does not render the device unusable in all or the majority of applications.
3	Features that were not the originally intended behaviour but should not cause any problems in applications.

Errata for the ARM-based embedded processor are grouped by module:

- A.1—the ARM920T AHB wrapper
- A.2—the ETM9 trace module
- A.3—the ARM922T processor core

A.1 Errata for the ARM920T AHB Wrapper Module

This section documents errata in the ARM920T AHB wrapper module, which is used in the Excalibur device.

A.1.1 Linefill Counter—Category 2

The burst counter for a cache linefill is incorrectly pre-loaded if the linefill follows a waited access. This results in 9 reads being performed on the AHB bus rather than the required 8. The ARM920T receives the correct data: the only effects of the extra read are that an extra cycle is taken up on the bus, and any side-effects are due to the addressed area being read-sensitive (e.g. a FIFO).

Work Around

The software workaround is to ensure that only address areas which are not read-sensitive are cached.

A.1.2 Error Response—Category 2

There is a bug in the ERROR response functionality in the wrapper. An ERROR response should only be accepted by the ARM920T if it is in response to a non-cacheable or non-bufferable access. The AHB wrapper only adds wait states to non-bufferable writes when ERROR support is added, since buffered writes do not need to be held up in order to propagate the respons. However, the ERROR response from the AHB is still propagated to the ASB in all cases. Thus a C/B access to the AHB which results in an ERROR response, followed by a NC/NB* access, means that the ERROR response appears on the ARM920T ASB interface as the response to the NC/NB access and so is accepted.

Work Around

The software workaround is to never access address regions which are capable of generating error responses with C/B accesses.

---

*NC Non-cacheable
NB Non-bufferable

A.2 Errata for the ETM9 Trace Module

See Table 1 on page 7 and associated text for an explanation of categorisation.

A.2.1 Invalid data trace following FIFO overflow—Category 1

Note: ARM has updated the errata classifications. This is a category 1 erratum under the previous definition, which was:

'Features which it is impossible, or very hard, to work around and are likely to affect use of this device.'

Description

The *Embedded Trace Macrocell* (ETM) contains a small FIFO which under some circumstances can overflow, leading to the loss of trace. This is normal behavior.

However, if the FIFO overflows during an extended wait-state period while attempting to capture data trace from a block data transfer instruction, then the data trace can become invalid.

When trace is resumed:

- Data values traced may be invalid until the next indirect branch.
- Data addresses traced may be invalid until the next trace gap, or the next periodic synchronization point.
- Instruction trace is unaffected.

An example of how exactly the data trace becomes invalid is shown in Table 2 on page 10. Note that, for simplicity, data address tracing is ignored.

Table 2. Example of invalid data trace following overflow during wait-state period

Address	Instruction	Data trace entering ETM FIFO[a]	Instruction flow reported by ETM	Data transfer reported by ETM	Notes
1004	LDMIA r6, {r0 – r4}	[r6] [r6+1] [r6+2]	1004 with data	r0 <= [r6] r1 <= [r6+1] r2 <= [r6+2]	ETM begins to log the data transferred from the contents of r6 to each registers in the list in turn, with r6 post-incremented each time.
					Wait-state begins, stalling data transfers. Overflow occurs (due to data trace continuing to enter the FIFO from the ETM's internal pipeline) causing tracing to be suspended
					FIFO drains and overflow clears
					Wait-state ends and data transfers resume
		[r6+3] [r6+4]	Overflow 1004 with data	r0 <= [r6+3] r1 <= [r6+4] r2 <= [r8][b] r3 <= invalid data[b] r4 <= invalid data[b]	The instruction is reported once more by ETM as trace restarts, data trace resumes and now incorrect data values are reported. The required operation was that no data should enter the FIFO at this point, as there is no way to indicate which transfers were traced.
1008	LDR r7, [r8]	[r8]	1008 with data	r7 <= invalid data[b]	The ETM now also reports an incorrect data value for register r7 The required operation was that the next operation after overflow would receive correct data trace. So, in this example: r7 <= [r8]

Notes:
(a) Corresponds to data transferred by the core
(b) Data from a future instruction

Note that, in this example, all the data transfers were reported either before or after the overflow. However, this may not necessarily be the case, and some transfers before the overflow, and the instruction itself, may not be reported,

Conditions

Extended wait-state periods can be caused by cache misses in cached systems, or the use of a slow memory system. Consequently, the problem is unlikely to occur in uncached systems or in systems where the speed of the memory is close to that of the processor. Since the clock speed on the ARM920T and ARM922T processors is slowed to that of the memory when a cache miss occurs, memory speed is not an issue on these processors, and consequently this erratum is less likely to occur on systems with these devices.

The block data transfer instruction must be executing when the overflow occurs. The instructions which fall into this category are as follows:

- ARM instructions:

  LDM, STM, SWP, SWPB, LDC, STC, LDRD, STRD, MCRR, MRRC.

- Thumb instructions:

  POP, PUSH, LDMIA, STMIA.

The wait-state period must begin before the overflow occurs. It is possible for an overflow to occur after the wait-state has begun due to trace continuing to enter the FIFO from the ETM pipeline. The wait-state period must continue until after the ETM has recovered from the overflow, having drained its FIFO of all pending trace.

The problem is more likely to occur with a small FIFO than a large FIFO for two reasons:

- A smaller FIFO is more likely to overflow.
- Once an overflow has occurred, a smaller FIFO takes less time to drain. Consequently, the length of the extended wait-state period required for the problem to occur is reduced.

As a result, the problem is most likely to occur with the small ETM configuration, and least likely with the large ETM configuration.

The minimum number of cycles required to drain the FIFO, and therefore the theoretical minimum length of the wait-state period, is shown in Table 3 on page 12.

Table 3. Minimum length of the wait-state period for the problem to occur

ETM Configuration	FIFO size	Minimum FIFO depth at overflow	Port size	Cycles to drain FIFO	Minimum number of consecutive wait states
Large	45 bytes	37 bytes	16-bit	19	20
			8-bit	37	38
			4-bit	74	75
Medium	18 bytes	10 bytes	16-bit	5	6
			8-bit	10	11
			4-bit	20	21
Small	9 bytes	1 byte	8-bit	1	2
			4-bit	2	3

While the minimum number of consecutive wait states required for this erratum to occur is increased by reducing the port size, a wider port size is still recommended as a more narrow port size will cause overflows to occur more frequently.

These cycle calculations are based on having eight free bytes in the FIFO when nine bytes are generated in a cycle. This occurs for the first data transfer after trace has been turned on, and requires both data value and data address tracing to be enabled. Most overflows will require a longer wait-state than this.

The ETM FIFOFULL signal attempts to preemptively insert processor wait states to prevent the FIFO from overflowing. Consequently, this means that this erratum will occur more often when FIFOFULL is enabled. In this case this erratum interacts with a separate category 3 erratum, "FIFOFULL LOW for one cycle during overflow—Category 3", documented on page 25. This causes FIFOFULL to be low for the first cycle of overflow, during which the ETM will not cause the processor to stall. As a result, while enabling FIFOFULL can increase the chance of this erratum occurring, it does not on its own cause the erratum to occur, because a wait-state must occur at the same time that FIFOFULL goes low for this one cycle.

Implications

While instruction trace remains unaffected, the user is unable to ascertain whether the data trace is correct.

The erratum must be considered when the following occur together:

- An overflow is reported.
- The first instruction traced following the overflow is a block data transfer instruction.
- The first instruction traced following the overflow includes data trace.
- Either:
  - The address of the instruction traced both before and after overflow is the same
  - The address of the instruction traced after overflow is the next instruction address that would have been expected had the overflow not occurred.

  Examples of this case are:

  - The instructions before and after overflow are both LDMIA instructions at address 1000.
  - The instruction before overflow is a branch at address 1010 that failed its condition codes, and the instruction after overflow is an STMDB at address 1014.
  - The instruction before overflow is an LDR to the pc at address 1020 which caused a branch to address 1040, and the instruction after overflow is an LDCL at address 1040. This case may be difficult to detect by the user, although it is possible for tools to be able to detect this because the target address of a branch is always traced, even if the next instruction is not.
- The amount of data trace lost depends on the position of the next indirect branch, and the data tracing mode selected. An indirect branch is any branch which is not a B, BL or BLX instruction, such as an LDR to the program counter.

One of the following apply:

The first instruction traced following the overflow is an indirect branch.

The data (addresses and values) traced with this instruction must be treated as invalid, but data traced for other instructions will be unaffected. It is always possible for invalid trace to be detected by the tools when the first instruction is an indirect branch, except in the case of an SWPB instruction when a 16-bit trace port is in use. If you are using tools which detect this, such as ARM's Trace Debug Tools, the data trace can be treated as valid if the first instruction following the overflow is an indirect branch and no error has been detected.

The next instruction to have data traced does not occur until after the next indirect branch.

The data traced with the first instruction must be treated as invalid, but data traced for future instructions is unaffected. It is not always possible for the tools to detect the error.

Other instructions have data traced before the next indirect branch, and only data addresses are being traced.

The data addresses must be treated as invalid for the first instruction, but will be valid for the instructions which follow.

Another instruction has data traced before the next indirect branch, and either only data values are being traced, or the first instruction following the overflow is an MRRC or MCRR.

The data values traced with all the instructions up to the next indirect branch must be treated as invalid. Data values traced after the next indirect branch are unaffected by this erratum and will be valid

Another instruction has data traced before the next indirect branch, and data values and addresses are both being traced.

The data values and addresses traced with all instructions before the next indirect branch must be treated as invalid. Data addresses traced for instructions after the next indirect branch, but before the next full 32-bit data address, must also be treated as invalid. The first data address output after each trace gap is a full 32-bit address, after which a full data address is forced if one has not been output for 1024 cycles.

While most tools do not report when a full data address has been output, if a data address differs from the previous data address traced in bits [31:28], then a full data address would have been output, allowing full data addresses that have not been periodically forced to be detected by the user. When a full data address cannot be detected in this way the user must either find the first data address that is at least 1024 cycles after the first instruction traced following the overflow, or treat all data addresses as invalid until the next gap in the trace.

Workaround

This workaround is for tool vendors only and must be read in conjunction with the *Embedded Trace Macrocell Specification* (ARM IHI 0014).

Development tool vendors can implement the above checks automatically in the trace tools, so that all data given to the user is guaranteed to be correct. In particular:

- The target address of branches is always known, so the comparison of the addresses before and after overflow can be accurate.
- The tools must detect when invalid trace is caused by this erratum, and not cause instruction trace to be lost as a result.
- The tools can detect when the first instruction after overflow is an indirect branch (traced with a PIPESTAT of BD) and not an SWPB with a 16-bit trace port, and not treat the data trace as invalid unless the branch address is earlier than expected. An SWPB cannot be reliably detected in this way because of the 1-byte gap that is sometimes left on a 16-bit trace port to align instruction addresses.
- Where data addresses are invalid until the next synchronization point, the tools should treat data addresses as valid following the next 5-byte data address output instead.

There is no known way to prevent the erratum from occurring, other than increasing the amount of filtering performed, particularly by ViewData, to reduce the chance of an overflow occurring.

Implications of workaround

In some circumstances substantial data trace, and in particular data addresses, might be lost. Data might be treated as invalid when it is not, particularly in systems where this erratum never or rarely occurs. As a result, it is recommended that if the workaround is implemented it is possible to turn it off.

A.2.2 Execution status unknown prior to an interrupt or prefetch abort—Category 2

Description

The trace port protocol allows for each instruction traced to be reported with a corresponding branch address to indicate the address of the next instruction. If this branch address is not output then the trace tools must calculate the address of the next instruction. If a branch address is output then the PIPESTAT pins of the trace port will indicate *Branch Executed* (BE) or *Branch with Data* (BD).

When an interrupt or prefetch abort occurs, the instruction which follows the last instruction to be executed should be traced (even though it did not execute) with a branch address to the relevant exception vector. In the case of interrupts this is often referred to as the interrupted instruction. The trace tools must recognize that the branch was to an interrupt or prefetch abort exception vector, and correctly treat that instruction as having not executed. In this way, the trace can reliably indicate if the last instruction executed failed its condition codes.

Sometimes the ETM does not trace this extra instruction but instead traces the last instruction executed as having branched to the exception vector.

Table 4 shows an example of incorrect trace caused by this erratum. Table 5 shows an example where this erratum does not occur.

Table 4. Example of incorrect trace behavior on interrupt

Actual instructions executed		Correct behavior	Actual behavior	
Address	**Instruction**	**Trace generated**	**Trace generated**	**Decompressed trace**
		Trace starts, address `0x00001000`	Trace starts, address `0x00001000`	
0x00001000	ADD r1, r1, 1	Instruction executed	Instruction executed	0x00001000
0x00001004	B 0x00001020	Instruction executed, branch to `0x00001020`	Instruction executed, branch to `0x00000018`	IRQ!
(0x00001020)	(Not executed due to interrupt)	Instruction executed, branch to `0x00000018`[a]		
`0x00000018` . . .	*IRQ hander* . . `SUBS pc, r14, #4`	IRQ handler traced . . Instruction executed, branch to `0x00001020`	IRQ handler traced . . Instruction executed, branch to `0x00001020`	`0x00000018` . . .
0x00001020	…	…	…	0x00001020

Note:
(a) Extra instruction traced to indicate interrupt

Table 5. Example of correct trace behavior on interrupt

Actual instructions executed		Actual behavior	
Address	**Instruction**	**Trace generated**	**Decompressed trace**
		Trace starts, address `0x00002000`	
0x00002000	ADD r1, r1, 1	Instruction executed	0x00002000
0x00002004	`BNE 0x00002020` Failed its condition codes	Instruction executed but failed its condition codes	`0x00002004` Failed its condition codes
(0x00002020)	(Not executed due to interrupt)	Instruction executed, branch to `0x00000018`[a]	IRQ!
`0x00000018` . . .	*IRQ hander* . . `SUBS pc, r14, #4`	IRQ handler traced . . Instruction executed, branch to `0x00002020`	`0x00000018` . . .
0x00002020	…	…	0x00002020

Note:
(a) Extra instruction traced to indicate interrupt

Conditions

The conditions under which this erratum occurs are not easily predicted. It never occurs if the last executed instruction failed its condition codes.

Implications

The last instruction executed before an interrupt or prefetch abort might be missing from the trace. Some tools always report the extra instruction traced as having executed, in which case the user will instead see an extra instruction reported before an interrupt or prefetch abort when this erratum does not occur.

If tracing continues until after the exception handler returns, the user can use the address of the first instruction executed upon returning from the exception to determine the actual behavior.

Workaround for tools vendors

Since the execution status is unknown, development tools cannot reliably report the last instruction executed before an abort or an interrupt.

The development tools must display the last instruction traced with its execution status shown as ‘unknown’, rather than suppress it as defined in the trace port protocol.

Implications of workaround

It is not possible to determine which instruction was executed immediately before an interrupt or prefetch abort, without referring to the last instruction traced before the interrupt.

A.2.3 Instruction displayed before and after overflow—Category 2

Description

When the FIFO overflows, the last instruction to be traced immediately before overflow may be repeated on recovery from overflow.

Conditions

The instruction must be executing when the overflow occurs and remain so until trace is re-enabled following overflow. This generally only arises if a large number of wait states occur. Consequently it should only occur in cached systems during a cache miss, or when using a very slow memory system.

Since the ETM `FIFOFULL` signal attempts to preemptively insert processor wait states to prevent the FIFO from overflowing, having FIFOFULL enabled can also increase the occurrence of this problem.

Implications

It can appear that an instruction is executed twice when it was only executed once.

Workaround for tools vendors

The development tools should be modified to discard the instruction before the overflow, only displaying the instruction after overflow when the addresses of the instructions before and after overflow match. Occasionally this would cause an instruction to be discarded unnecessarily, but a much larger number of instructions would already have been discarded due to the overflow.

It is important that the instruction before overflow is the one to be discarded, and not the one following overflow recovery, in case the erratum is falsely detected. If trace is turned off during overflow (TraceEnable goes LOW), then enabled again (TraceEnable goes HIGH) some time after overflow recovery, the first instruction to be traced following overflow may be significant and must be traced.

Implications of workaround

An extra instruction will occasionally be lost during an overflow occurring in a loop. This will not be detectable by the user.

A.2.4 Address range cannot include 0xFFFFFFFF—Category 2

Description

There is no way to define a range which includes `0xFFFFFFFF`. Ranges are defined to be exclusive of the upper address, so a range with an upper address of `0xFFFFFFFF` only includes addresses up to `0xFFFFFFFE`.

Conditions

All.

Implications

The ability to monitor accesses to this address is seriously reduced.

Workaround

To monitor accesses to `0xFFFFFFFF`, the upper address on the range must be set to `0xFFFFFFFF`, with the size mask set to clear bits 1:0 (bits 4:3 of the access type register must be set to *b11*). This causes the upper bound of the address range never to match, and so the range has no upper limit and accesses to `0xFFFFFFFF` are monitored correctly.

Implications of workaround

No implications.

This will become part of the ETM specification.

A.2.5 Extra CPRT data traced—Category 2

Description

The ETM specification states that CPRT data (data transfers between ARM processor registers and coprocessor registers, which are caused by the instructions MRC, MCR, MRRC, and MCRR) must be traced if and only if the MonitorCPRT bit is set (bit 1 of register `0x00`, the control register).

Under some circumstances CPRT data can be traced when this bit is not set.

Conditions

This occurs when both of the following are true:

- ViewData is enabled.
- Either data value or data address tracing are enabled (bits 3:2 of register `0x00`, the control register).

Implications

Extra CPRT data is traced. Since the number of such transfers is relatively small, the amount of extra data generated should not be significant.

As there is no requirement for the tools to correctly handle CPRT data traced when the MonitorCPRT bit is not set, this may result in unexpected behavior.

Workaround

If this erratum causes an error in the development tools, the user must always enable coprocessor register transfer tracing.

The tools must be able to handle CPRT data traced even when no CPRT data was expected.

If this the extra trace is deemed to be confusing to the user, the tools can ignore CPRT data when CPRT data tracing has been disabled.

Implications of workaround

A small amount of bandwidth may be wasted.

A.2.6 Stall cycles reported on wrong instruction—Category 2

Description

The timestamps or cycle numbers, if captured, might not truly reflect the number of cycles taken by each instruction. Stall cycles which cause an instruction to take longer than the minimum time to execute might be reported on an earlier instruction.

It is occasionally useful to be able to determine the exact cycles on which instructions are executed, for example to perform basic performance analysis or to diagnose problems with the memory system. The information can be preserved by one of two methods:

- By selecting *cycle-accurate mode*, the ETM can be configured to cause trace to be captured on every cycle during a trace region.
- By using a trace port analyzer or logic analyzer which is capable of saving timestamps with each cycle of trace.

Under some circumstances, extra instruction execution cycles are reported alongside a previous instruction, making it appear that that instruction took longer to execute instead. While the number of cycles that a sequence of instructions takes to execute is correctly reported, the number of cycles taken by individual instructions is not.

This erratum does not affect ETM9 Rev 0a, although ARM strongly recommends that the latest revision of ETM9 is used in all new designs. The erratum was introduced as a result of improvements to the effectiveness of the `FIFOFULL` mechanism, although it is occurs whether `FIFOFULL` is enabled or not.

Table 6 shows an example where a stall caused by an LDR is traced with the wrong instruction. Note that the ETM reports an instruction when the following instruction begins execution. Therefore the time taken by an instruction is the time between that instruction and the previous instruction in the trace.

Table 6. Example of correct and incorrect trace in the presence of stalls (Part 1 of 2)

Cycle	Address	Instruction	Actual trace	Expected trace	Notes
500	1000	NOP			Instruction enters ETM pipeline
501	1004	NOP			1004 triggers tracing of 1000, 9 cycles later
502	1008	NOP			
503	100C	NOP			
504	1010	NOP			
505	1014	NOP			
506	1018	NOP			
507	101C	LDR			
508		(LDR stalled)			Most of ETM pipeline stalled along with processor pipeline
509	1020	NOP			
510	1024	B 1040	IE (instruction executed) for 1000	IE for 1000	

Table 6. Example of correct and incorrect trace in the presence of stalls (Part 2 of 2)

511		(Branch delay)	WT (wait)	IE for 1004	Stall in ETM pipeline observed after only 3 cycles
512		(Branch delay)	IE for 1004	IE for 1008	
513	1040	NOP	IE for 1008	IE for 100C	
514			IE for 100C	IE for 1010	
515			IE for 1010	IE for 1014	
516			IE for 1014	IE for 1018	
517			IE for 1018	WT	Stall should be observed in line with other instructions
518			IE for 101C	IE for 101C	
519			IE for 1020	IE for 1020	
520			WT	WT	Stalls caused by the processor are correctly reported
521			WT	WT	
522			IE for 1024	IE for 1024	Instruction traced upon completion

Conditions

As shown in Table 6, only external stalls that are caused by deasserting CLKEN (such as memory stalls) are misreported. All stalls caused by the processor, including branch delays, register interlocks and coprocessor busy-waits, are reported correctly. External (CLKEN) stalls are reported 6 cycles early.

Implications

It is hard to use the timestamp information to gain information on stalls caused by the memory system. While this is beyond the scope of the original design aims of the ETM, precise stall information has proven to be extremely useful to some users in debugging system level issues.

Workaround

It is often possible to model the behavior of the core as it would behave if there were no external stalls to determine which stalls are external and which are internal. To do this in all situations would require a complete cycle-accurate model of the core. However, in most cases the number of cycles required by each instruction can be easily predicted, and is documented in the Technical Reference Manual for the core.

It is therefore possible to determine where external stall cycles have been inserted, and to count forward 6 cycles from that point to find where they should have been inserted, not counting other external stall cycles. This is a complex procedure which cannot easily be automated, and cannot be regarded as a complete workaround.

Table 7 shows an example of this technique. Note that, as in Table 6, the trace reports stall cycles before the corresponding instruction is traced, not after.

Table 7. Example of how to calculate the correct location of a stall (Part 1 of 2)

Cycle	Address	Instruction	Actual trace	Reconstructed trace	Notes
600	2000	MOV r1,r0			
601	2004	LDR r2,var0			
602		(interlock on r2)			This is an internal stall and can be predicted
603	2008	ADD r2,r2,#1			
604	200C	B 2018			
605		(Branch delay)			These cycles are also predictable internal stalls
606		(Branch delay)			
607	2018	ADD r2,r2,#1			
608	201C	LDR r3,var1			
609		(LDR stalled)			An external stall
610	2020	LDR r4,var2	IE (instruction executed) for 2000	IE for 2000	
611	2024	ADD r2,r2,#1	WT (wait)	WT	Only one stall was predicted for 2004. One must be an external stall. Counting forward 6 from the first puts it on 2018, the second on 201C. Remove it and try to place later.
612			WT	IE for 2004	
613			IE for 2004	IE for 2008	
614			IE for 2008	WT	
615			WT	WT	
616			WT	IE for 200C	
617			IE for 200C	IE for 2018	The stall could correspond to 2018 in theory, but in this system only the memory system can cause external stalls, and the code is in fast instruction memory, so an ADD could not cause an external stall.

Table 7. Example of how to calculate the correct location of a stall (Part 2 of 2)

618			IE for 2018	WT	The stall is far more likely to correspond to 201C, as this causes a LDR to on-chip memory. We can therefore deduce that this LDR caused precisely 1 stall cycle.
619			IE for 201C	IE for 201C	
620			IE for 2020	IE for 2020	The stall could not correspond to this load, as it is not 6 cycles ahead of a detected external stall in the trace.
621			IE for 2024	IE for 2024	Note that the total number of cycles taken for the sequence of instructions is reported correctly.

Note: If 2018 is also an LDR, rather than an ADD, then it would not be possible to determine which of the two instructions 2018 or 201C caused the stall.

A.2.7 FIFOFULL LOW for one cycle during overflow—Category 3

Description

Although FIFOFULL correctly becomes asserted when the space available in the FIFO is less than the minimum value specified in the FIFOFULL Level register of the ETM (register `0x0b`), it incorrectly becomes de-asserted for a single cycle when the FIFO overflows.

Conditions

This occurs for one cycle, on the same cycle in which the FIFO overflows.

Implications

One extra cycle of trace can be lost during FIFO overflow. Since this case only occurs when overflow is already about to occur, the ability of FIFOFULL to prevent overflow is unaffected.

Workaround

None required. The erratum causes the loss of an extra cycle of trace when there is an overflow, so the effect is not significant.

Implications of workaround

No implications.

A.2.8 Extra instruction traced prior to debug entry—Category 3

Description

An instruction selected as a breakpoint may be traced prior to debug entry, even though it was not executed. It is flagged as having failed its condition codes, regardless of whether it is conditional.

Conditions

This has been observed in devices based on the ARM9TDMI (for example the ARM9TDMI, ARM920T, and ARM922T cores) only. It has not been observed in devices based on the ARM9E (for example the ARM9E-S, ARM966E-S, and ARM946E-S cores).

Implications

It will appear to the user that the breakpoint occurred at the wrong time.

Workaround

No workaround is known.

Implications of workaround

No implications.

Correction

No correction is planned.

A.3 Errata for the ARM922T Processor Core

A.3.1 LDM of user mode registers (ARM9TDMI–8)—Category 2

ARM9 Bug tracking database entry : CPC00_CAM_000013

Summary

Under specific conditions, a LDM to user mode registers will not operate correctly. These instructions take the form:

```
LDM{<cond>}<addressing_mode> <Rn>,<registers_without_pc>^
```

These instructions are only used in system code and not in application code.

Description

A LDM to user mode registers performs a load to the user mode registers whilst the processor is in a privileged mode.

Under specific conditions (see below), this instruction will fail to operate correctly. This results in not all the registers in the register list being written correctly. It may also cause further failures, dependent on the construction of the memory system, since the data memory request signals are driven in an incorrect manner in the failing situation.

Example of failing instruction

```
LDM sp,{sp,lr}^
```

Conditions

This errata exists in the following circumstances:

A LDM to user mode registers, where:

1. The base register is register 8 or greater and
2. The base register is the first register (lowest register number) in the register list and
3. There is more than one register is in the list.

Notes:

In all privileged modes this errata exists if the base register is 8 or greater. This is not restricted to FIQ mode.

Instructions of the form

```
LDM{<cond>}<addressing_mode> <Rn>,<registers_and_pc}^
```

operate correctly since these are not LDM to user mode register instructions. These instructions perform a return from exception.

Example of failing instructions

```
LDMIA sp,{sp,lr}^
LDMIA r8,{r8-r10}^
```

Examples that do NOT fail

```
LDMIA sp,{r8,sp,lr}^ ; Ok, since first register is not the base register
LDMIA r5,{r5,sp,lr}^ ; Ok, since base register is < r8
LDMIA sp,{sp}^       ; Ok, since only one register in the list
LDMIA sp,{sp,lr,pc}^ ; Ok, since PC in the list and hence is a return from
                     ; exception instruction and not a load of user mode
                     ; registers.
```

Implications

This errata only applies to hand crafted assembler code, as the ARM compiler does not generate such instructions. The use of this instruction is typically limited to a few places in exception handlers, thus limiting the scope of this erratum.

Work-around

This instruction is only used by hand crafted assembler code. The ARM compiler will not generate this instruction. To work-around this errata the LDM should be split into two separate instructions.

e.g. The instruction:

```
LDMIA sp,{sp,lr}^
```

should be split into

```
LDMIA sp,{sp}^
LDMIA sp,{lr}^
```

and

```
LDMIA r8,{r8-lr}^
```

should be split into

```
LDMIA r8,{r8}^
LDMIA r8,{r9-lr}^
```

A.3.2 Debug Request coincident with Pipeline Hazards (ARM9TDMI–9) —Category 2

Summary

The processor may return to normal program execution at an incorrect point if **EDBGRQ** or the scan chain created debug request is asserted whilst it is performing certain tasks.

Conditions

There are three scenarios in which this erratum can occur:

1. Debug Request occurs whilst the processor is waiting for a coprocessor instruction to be completed by a coprocessor or
2. Debug Request occurs whilst the recognition of a Data Abort is in progress or
3. Debug Request occurs whilst the recognition of an instruction causing a watchpoint is in progress.

☞ Recognition of a Data Abort is said to be in progress if the last memory access asserted the DABORT signal, but the processor has not yet begun execution at the Data Abort vector.

Recognition of a watchpoint is said to be in progress if the last memory access caused a watchpoint, but debug entry has not yet completed.

If the above conditions are met then the debug entry mechanism fails to behave in the defined manner and the device may return from debug and execute from an incorrect address.

Implications

For each scenario the following implications are expected given the above conditions:

Busy-waiting Coprocessor Instructions

In this form the debug request supersedes the currently executing coprocessor instruction, which would normally not occur. As such the PC is not the correct value as debug entry proceeds. Correspondingly, on return from debug the standard return address calculation produces an incorrect address and thus unintended or unpredictable device behaviour may result.

Data Aborts

In this form the debug request causes correct debug entry and exit. However, the link register address calculation for the return from the Data Abort handler is incorrect. Correspondingly, on return from the Data Abort handler unintended or unpredictable device behaviour may result.

Watchpoints

In this form the debug request causes debug entry first, but the watchpoint is still pending and has yet to execute the next instruction before it takes full effect. As debug entry has already occurred this next instruction will be the first instruction within debug mode to be executed. After this instruction executes, debug entry occurs for a second time, even though the device is already in debug, and results in the premature exiting of debug. Consequently, the return from debug is to an incorrect address and thus unintended or unpredictable device behaviour may result.

Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

A.3.3 Data Abort and Watchpoint with Breakpoint Following (ARM9TDMI–10)—Category 2

Summary

The processor may fail to execute the abort handler if a data abort occurs on a watchpointed instruction and a breakpoint is in the execution pipeline.

Conditions

The conditions for this erratum are:

1. An instruction that causes both a Data Abort and a watchpoint to occur and
2. The execution of the following instruction will cause a breakpoint to occur

There should be no data dependency between the two instructions such that a pipeline interlock occurs. If there is such data dependency then the processor behaves correctly.

If the above conditions are met then the Data Abort may be missed.

Implications

In this erratum Data Abort entry is halted by debug entry and this causes the state indicating a Data Abort to be lost on return from debug. Hence the Data Abort handler will fail to be invoked. This may result in unintended or unpredictable device behaviour.

Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

A.3.4 Watchpoint coincident with Debug Request (ARM9TDMI–11)—Category 2

Summary

The processor can falsely exit debug state and continue execution if **EDBGRQ** or the scan chain created debug request is asserted whilst debug entry is occurring.

Conditions

A combination of two conditions is required to cause this erratum:

1. Debug request is asserted and
2. An instruction that generates a watchpoint is executing

If the above conditions are met then the debug entry mechanism fails to behave in the defined manner and the device may return from debug and execute from the incorrect address.

Implications

In this erratum, the debug request takes affect before the complete recognition of the watchpoint. This may result in unreliable debug entry, the watchpoint being missed and premature exit of debug state. Thus unintended or unpredictable device behaviour may result.

Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

A.3.5 Register controlled shift data operations where the destination is the PC (ARM9TDMI–1)—Category 3

ARM9 Bug tracking database entry : CPC00_CAM_000001

Summary

A data operation with a register controlled shift to the PC may calculate an incorrect result

Description

This fault is exhibited by any data operation involving a register-specified shift with the Program Counter as the destination. This is in effect a calculated branch, with an unusual branch address calculation. The branch target address calculated by the instruction is incorrect.

The ARM C compiler will not generate this instruction. Other compilers are extremely unlikely to produce this instruction, as no standard high level languages source code constructs which would map to this instruction.

It is also extremely unlikely that this instruction has been used in assembler code, as is explained below.

Conditions

Exists for any instruction that involves a register-specified shift or rotate operation with the PC as the destination for the resulting data. The fault does not occur for an immediate specified shift or rotate to the PC.

Implications

Data operations involving a register controlled shift and where the destination register is the PC have an unpredictable behaviour on an ARM9TDMI processor core and any processor containing an ARM9TDMI; for example ARM920T.

The current ARM compiler cannot generate instructions of this class and so this would only be encountered in hand coded assembler. ARM's software staff have considered possible uses for such an instruction in assembler code, and have only found one theoretical use for this instruction, which is a strange branch table described below.

For these reasons this erratum is not expected to cause any restrictions or problems in using the ARM9TDMI.

Examples:

```
MOV(S)   PC, Rm, <SHIFTOP> Rs
ADD(S)   PC, Rn, Rm, <SHIFTOP> Rs
```

The only theoretical use for these instructions which has been suggested would be for a branch table that was organised in powers of 2.

```
MOV                  r0,#0x100
MOV                  r1,#0x4
ADD                  PC,r0,r1, LSL r2
0x1004:              Code seq1 : 1 instruction
0x1008 -> 0x100C:    Code seq2:2 instructions
0x1010 -> 0x101C:    Code seq3:4 instructions
0x1020 -> 0x103C:    Code seq4:8 instructions
0x1040 -> 0x107C:    Code seq5:16 instructions
0x1080 -> 0x10FC:    Code seq6:32 instructions
0x1100 -> 0x11FC:    Code seq7:64 instructions
0x1200 -> 0x13FC:    Code seq8:128 instructions
```

To date ARM knows of no examples of an application for a branch table organised in such a manner. However, if one was required then an alternative code sequence could be used.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com