ARM1022E Rev0 Errata List

CPU Cores Division

Document number:	ARM1022E -PRDC-000705 12.0
Date of Issue:	24th June 2002
Author	Paul Garden
Authorised by	John Cornish

Abstract

This document describes the known errata in the all ARM1022E revisions.

Keywords

Errata, bug, workaround, ARM1022E.

Contents

1 ABOUT THIS DOCUMENT

1.1 Change History

Issue	Date	Change
0.1	9/6/2001	First Draft for Domino.Doc
1.0	9/28/2001	First Release on Domino
2.0	9/28/2001	Update to document title
3.0	10/10/2001	Changed header to 1022E
4.0	5/1/2002	Added SMULxy errata Added AHB HTRANS errata
5.0	6/25/2001	Changed title on Domino
6.0	6/25/2002	Added CP15 LT bit errata
7.0	9/12/2002	Added Breakpoint and Incorrect Halting erratas
8.0	10/24/2002	Added Qflag Errata
9.0	5/13/2003	Added Erroneous BIU read
		Added unable to entry Debug state
10.0	5/20/2003	Updated the English in above errata entry
11.0	5/21/2003	Changed minor error in Abstract
12.0	6/17/2003	Added r0p2 into table to show cat 1 errata fix

1.2 References

This document refers to the following documents.

Ref.	Document No	Author(s)	Title

1.3 Scope

This document describes the errata discovered in the implementation of the ARM1022E Rev 0.0, categorised by level of severity. Each description includes:

- where the implementation deviates from the specification
- the conditions under which erroneous behaviour occurs
- the implications of the erratum with respect to typical applications
- the application and limitations of a 'work-around' where possible
- the status of corrective action.

1.4 Terms and Abbreviations

This document uses the following terms and abbreviations.

Term	Meaning

2 CATEGORISATION OF ERRATA

Errata recorded in this document are split into three groups:

Category 1 Features which are impossible to work around and severely restricts the use of the device in all or the majority of applications rendering the device unusable.

Category 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.

Category 3 Features that were not the originally intended behaviour but should not cause any problems in applications.

2.1 Errata Summary

The ARM1022E r0p2 is the current revision available and has known errata, as indicated by the table in the section below.

2.2 Product Revision and Errata Summary Table

The errata associated with this product are categorised in the following way.

Errata Description	Rev r0p0	Rev r0p1	Rev r0p2
Category 1			
3.1) Erroneous Data Bus Interface AHB read preceding a buffered write/data cache eviction sequence in all HCLK:GCLK ratios	**Yes**	**Yes**	**No**
Category 2			
4.1) Erroneous Q Flag Value for Enhanced DSP Multiply-Accumulate Following Bounced Coprocessor Instructions/Hardware Watchpoints	**Yes**	**Yes**	**Yes**
4.2) Breakpoint Instructions and Branch Prediction in Halt (Hardware) Debug Mode	**Yes**	**Yes**	**Yes**
4.3) ARMV5TE SMULxy multiply instruction failure under special conditions	**Yes**	**No**	**No**
4.4) Incorrect halting (hardware) debug interaction with exceptions	**Yes**	**Yes**	**Yes**
4.5) Unable to enter debug state on EDBGRQ or JTAG HALT in a predicted branch-to-self loop	**Yes**	**Yes**	**Yes**
Category 3			
5.1) AHB Compliance Problem With HLOCK Signal	**Yes**	**Yes**	**Yes**
5.2) AHB AMBA Rev 2.0 Compliance Problem With HTRANS	**Yes**	**No**	**No**
5.3) CP15 control register 1 LT bit set causes erroneous behaviour	**Yes**	**Yes**	**Yes**
5.4) DFT: HRESPI[1], HESPI[0], HRESTP[1], HRESPD[0] input ports do not have wrapper cells	**Yes**	**Yes**	**Yes**

3 CATEGORY 1 ERRATA

3.1 Erroneous Data Bus Interface AHB read preceding a buffered write/data cache eviction sequence in all HCLK:GCLK ratios (r0p0, r0p1)

3.1.1 Summary

Under certain conditions, the processor may issue a data AHB read transaction, which does not represent any executed instruction. The read is an error in the processor behaviour and will result when the processor executes a load instruction, which misses and causes a data cache linefill, which generates an eviction/castout, followed by a buffered store. If the buffered store timing is correct, the erroneous data AHB read will result.

Note that the behaviour has been found to be present for all clock ratios (integer multiples) between the data AHB clock, HCLK, and the processor clock, GCLK.

3.1.2 Description

The conditions which generate the errata can be described using the following instruction sequence and one other sequence which will be described later.

LDR rX	* generates a cache linefill and castout/eviction
STR rY	* placed into the write buffer
(B rZ or NOP or LDR rZ or STR rZ)	* optional instruction

The correct behaviour of the executed instructions on the data AHB interface can be seen below, when excluding the optional instructions:

… -> LINEFILL rX -> IDLE -> EVICTION rA -> IDLE -> STORE rY -> …

In the diagram above the LDR rX instruction generates a cache linefill followed by a cache eviction at address rA. The processor will then detect the presence of the STR rY instruction in the write buffer. The data BIU will then issue a buffered write. Depending on the optional instruction, the processor may execute a non-load/store instruction (NOP, B rZ, other), it may execute a LDR rZ instruction (memory regions CB, CNB, NCB, or NCNB), or it may execute a STR rZ instruction (memory regions CB, CNB, NCB, or NCNB). The optional instruction executed, if a load or store which needs to access the data AHB interface, will then generate a transfer on AHB corresponding to the load or store.

If the erroneous behaviour arises, the processor will issue an additional data AHB load prior to the STR rY operation being presented onto the data AHB interface. This erroneous data AHB load can be to two distinct addresses depending upon the timing of the optional instruction with respect to the STR rY instruction, the processor clock ratio and the behaviour of the data AHB signals (HGRANTD, HREADYD, HRESPD[1:0], …).

It has been found that four possible scenarios can be generated which always results in an erroneous data AHB load transfer.

1. A load is issued from the processor onto the data AHB interface to address rY with the following data AHB outputs: HTRANS=NSEQ, HBURST=SINGLE, HPROT=attributes of STR rY, HWRITE=read, HSIZE=BYTE. The buffered write, STR rY, will be adjacent and following the erroneous load to address rY on the data AHB interface. The sequence is shown below which represents data AHB transfers:

   IDLE -> LOAD rY -> STORE rY -> IDLE

2. A load is issued from the processor onto the data AHB interface to address rY with the following data AHB outputs: HTRANS=NSEQ, HBURST=SINGLE, HPROT=attributes of STR rY, HWRITE=read,

HSIZE=BYTE. The buffered write, STR rY, will be not be adjacent, but following an IDLE transfer which follows the erroneous load to address rY on the data AHB interface:

IDLE -> LOAD rY -> IDLE -> STORE rY -> IDLE

3. A load is issued from the processor onto the data AHB interface to address rZ with the following data AHB outputs: HTRANS=NSEQ, HBURST=SINGLE, HPROT=attributes of LDR/STR rZ, HWRITE=read, HSIZE=BYTE. The buffered write, STR rY, will be adjacent and following the erroneous load to address rZ on the data AHB interface. The sequence is shown below which represents data AHB transfers:

   IDLE -> LOAD rZ -> STORE rY -> IDLE -> LOAD/STORE rZ

4. A load is issued from the processor onto the data AHB interface to address rZ with the following data AHB outputs: HTRANS=NSEQ, HBURST=SINGLE, HPROT=attributes of LDR/STR rZ, HWRITE=read, HSIZE=BYTE. The buffered write, STR rY, will be not be adjacent, but following an IDLE transfer which follows the erroneous load to address rZ on the data AHB interface:

   IDLE -> LOAD rZ -> IDLE -> STORE rY -> IDLE -> LOAD/STORE rZ

From the four possible scenarios, it can be noted that the erroneous load to either address rY or rZ is not a random address but a predictable address. It can also be noted that the erroneous load to address rY will always be to a region of memory that is cacheable. Further, it can be noted that the erroneous load to address rZ can be to any memory region: CB, CNB, NCB, or NCNB. The last thing to note is that the erroneous transfer will never be a store.

A new fifth case can also occur and is a result of the “optional” instruction being a branch or a MRC/MCR to coprocessor 15. In these cases, the mangling of the address, as will be described, will be presented on the data BIU interface and will be the address of the phantom load.

5. A load is issued from the processor onto the data AHB interface to address rZ with the following data AHB outputs: HTRANS=NSEQ, HBURST=SINGLE, HWRITE=read, HSIZE=BYTE. The buffered write, STR rY, may or may not be adjacent:

   IDLE -> LOAD rZ [-> IDLE] -> STORE rY -> IDLE

Using the code sequence below, the branch issue will be detailed to describe all operations that can possibly mangle the address to the data BIU.

```
Loop_41
      LDR     r5,[r1]          ; generate write data
      29 NOPs
      STR     r11,[r8]
      LDR     r7,[r12]         ;
      B  loop_41a              ;  continue
      NOP
Loop_41a
      STRB    r4,[r1],#0x20    ; write 1 byte and increment address by 32
      B  loop_41               ;  continue
```

There are two cases that have been identified and have been reviewed by ARM which cause the phantom read operation to result in a mangling of a branch target address and a previous load/store lookup. To discuss these, lets first review the issue of how the phantom read is constructed.

The phantom read can only be produced in the ARM1020E/ARM1022E macrocell due to a missing qualifier in the data AHB bus interface unit. The data BIU must be completing a cache eviction, as a result of a cache linefill, and at the end of that eviction a bufferable store must be placed into an empty write buffer. Upon completing the eviction on the data AHB interface, the macrocell will under certain conditions issue a read to some address (to be known as a phantom read). The phantom read due to AHB conditions may be adjacent to the buffered write from a bus transaction perspective. If it is not, the cycles between the phantom and the buffered write will always be IDLE transfer cycles.

The address of the phantom read is in question. Due to the severity of the bug (category 1), a design modification must be performed. However, ARM has been determining whether there is a possible software work-around for any partner which has a closed looped system (i.e. complete control over the software and hardware). To this end the conditions that have been defined are as follows in the software and hardware:

```
1a) all memory regions and page entries fall into two categories:
      a) write back
      b) non-cacheable non-bufferable (NCNB)
2a) any read destructive regions of memory must be safe guarded due
    to the phantom read corrupting that region.
3b) read destructive regions only all into category NCNB.
```

With these restrictions, the expected address of the phantom read should be restricted to load/store instructions; however, revisiting the load/store adder usage, the load store unit can be used to calculate addresses for the following set of instruction:

```
1b) LDR/STR
2b) LDC/STC
3b) LDM/STM
4b) LDRD/STRD
5b) PLD
6b) SWP
7b) MRC/MCR to coprocessor 15
8b) B,BL,BX,BLX (branches)
9b) MOV RC, Rx, <sh> #0
```

The instructions of interest are items 7b, 8b, and 9b. These are non-load/store operations. Item 7b is an interesting case since the operation is to coprocessor 15. The instructions that utilize the load/store adder are shown below, with rX being the register used to hold the address for the coprocessor 15 instruction:

```
MCR/MRC p15,0,rX, c7,cY,Z => CP15 cache ops
MCR/MRC p15,0,rX, c8,cY,Z => CP15 MMU ops
MCR/MRC p15,0,rX, c9,cY,Z => CP15 cache lockdown control
MCR/MRC p15,0,rX,c10,cY,Z => CP15 MMU lockdown control
MCR/MRC p15,0,rX,c15,cY,Z => CP15 test ops
```

Items 8b and 9b are operations which use the PC of the machine, implying that the destination of the operation will be in a segment of code and not a read destructive location of memory. Note that in these particular cases, the load/store adder is used to calculate the target of a branch. Note that in the original instruction sequence described, the sequence of

```
LDR a -> BR x -> ... -> STR b -> BR y
```

is described. The branch target will get mangled with a historical load/store access which used the memory management unit, MMU, where the MMU is holding the historical access address and page information. So in the case of the access being to a 1MB section, the combined address would be a combination of the historical load/store access address bits [31:20] with the branch target address bits [19:2] and bits [1:0] being zeros. Note that the following combinations of address can be realized:

```
                    historical    branch   constant
                    load/store    target    zero
 1MB section  =>      [31:20]     [19:2]    [1:0]
64KB page     =>      [31:16]     [15:2]    [1:0]
16KB sub-page =>      [31:14]     [13:2]    [1:0]
 4KB page     =>      [31:12]     [11:2]    [1:0]
 1KB page     =>      [31:10]     [ 9:2]    [1:0]
```

In the example code shown above and in all cases due to timing, key word being timing, of the executed branch instructions relative to the bufferable store instruction, the phantom read address may be to the following and only the following:

```
1c) (B loop_41a) combined with (STR r4, [r1], #0x20)
2c) (STR r4, [r1], #0x20)
3c) (B loop_41) combined with (STR r4, [r1], #0x20)
4c) (B loop_41a) combined with (LDR r7, [r12])
5c) any load/store operation following (STR r4, [r1], #0x20)
```

Note that every scenario is 100% a function of timing and cannot be controlled by the processor during the execution of a particular code segment. This is due to the many conditions which arise. For example, when and how often do cache evictions occur, timing of linefills, clock ratio of core-to-bus, stalls on the bus, stalls in the processor pipeline, conditional execution of instructions, asynchronous events presented to the processor, etc.

To further analyse the issue, one must question whether item 5c above can be a mangling of a load/store operation with a branch target. The data BIU construction is such that it will block all subsequent addresses, excluding a bufferable store, and will cause the data BIU to stop sampling a new physical address. Therefore, the phantom can additionally be to the load/store address for item 5c. In the case of the buffered write for item 5c, timing will permit a mangled address to be presented for the bufferable store. Note that the address will always be limited to the same page as the bufferable store and, hence, will have the same memory characteristics as the bufferable store in terms of memory region type.

A possible software fix for a closed loop system is to do the following:

```
1d) determine in software and hardware all memory accesses which can cause
    a read of that memory to be read destructive. this may be a peripheral
    register as well.
2d) modify using the code template below all read destructive access as
    follows:

            < disable all interrupts >
            LDR rA, [rY]                ; non-cacheable load to an area
                                        ; of memory that is not read
                                        ; destructive
            LDR rB, [rZ]                ; read destructive access
            < enable all interrupts >
```

This fix will force the phantom's address, if phantom present, to become the address of item 2d's load to address rY. Note that this is not the read destructive address of items 2d's load to address rZ. Interrupts have to be disabled in order to remove the possibility of the asynchronous interrupt re-creating the behaviour.

3.1.3 Conditions

This problem can only occur if the processor has enabled the memory management unit, MMU, the data cache, and the write buffer. The MMU must have page entries programmed such that they support CB memory regions in order to produce a data cache eviction/castout.

3.1.4 Implications

If the erroneous data AHB load operation is to a read destructive region of memory, reading memory can cause loss of program data. An example of this is the reading of an interrupt vector register or a peripheral shift register. If the erroneous data AHB load is not to a read destructive region of memory, reading memory should cause no loss of program data.

An important point to note is that the erroneous data AHB load will never return the read data on HRDATAD or response of error on HRESPD to the processor. Hence, there will never be corruption of the register file or signalling of a data abort to the processor.

3.1.5 Workaround

Known solutions to the erroneous data AHB load are as follows:

- the MMU page tables can only be programmed to support NCNB, NCB, and CNB. CB must be converted to CNB.
- the write buffer must be disabled using CP15 register 1 bit 3, W bit. This will in turn cause all buffered stores to become non-buffered stores.
- software must safe-guard all read destructive operations by using the following sequence:
    1. execute the buffered store at rY
    2. disable all interrupts
    3. perform a NCNB load to a memory location that is not read-destructive
    4. perform the optional LDR/STR at rZ, where this operation is read-destructive
    5. enable all interrupts

All known solutions require a dramatic impact to the performance of the processor. Or the solution will require extreme modifications to software.

4 CATEGORY 2 ERRATA

4.1 Erroneous Q Flag Value for Enhanced DSP Multiply-Accumulate Following Bounced Coprocessor Instructions/Hardware Watchpoints (r0p0, r0p1)

4.1.1 Summary

Certain events that occur on an instruction that is followed by a DSP multiply-accumulate instruction that sets the Q flag (`SMLAWx` or `SMLAxy`) will show the Q flag set in the CPSR and SPSR_und upon entering the event handler. These events include bounced (thrown to software via the undefined instruction handler) coprocessor instructions, and watchpoints in hardware debug mode.

4.1.2 Description

If a coprocessor instruction (MCR/MRC/STC/LDC/CDP) is executed with the coprocessor number being 0 through 13, this coprocessor instruction is sent to external coprocessors. If the given coprocessor does not exist or rejects the instruction, the instruction is said to be BOUNCEed. Likewise, a CP14/15 coprocessor instruction can be internally rejected due to permission violations like attempting to run CP15 instructions in user mode or write certain CP14 registers in hardware debug mode.

In these cases, any instructions that are in the pipeline will be flushed and the undefined instruction handler will be entered. However, the flush of the DSP multiply-accumulate instruction is not completed correctly, and the resultant Q-flag value generated by this instruction is updated.

Additionally, if a load or store instruction attempts to read or write from an address that has a watchpoint set on it, and the debug is in hardware (halt) mode, and a DSP multiply- accumulate instruction follows the watchpointed instruction, the Q-flag will also be set before the watchpoint halts the processor.

The DSP multiply- accumulate instructions affected by this problem are: `SMLABB, SMLABT, SMLATB, SMLATT, SMLAWB`, and `SMLAWT`. The DSP add and subtract instructions are NOT affected.

4.1.3 Conditions

This problem can be encountered whenever there is a back-to-back sequence of these two classes of instructions. This problem may not be exhibited for all rejected CP14 or CP15 instructions.

Additionally, if branch prediction is enabled, the two instructions can be separated by a predicted unconditional or conditional branch and still exhibit the problem if the branch is predicted properly.

4.1.4 Implications

Under most program conditions, a BOUNCEd coprocessor instruction will be either emulated or cause an error condition. In the case of emulation, the coprocessor emulation code must appear transparent to the main program and therefore will save and restore the state of the processor before modifying any registers. Coprocessors cannot directly read the Q flag, so emulation code will not rely on the Q flag state of the initial program. The emulation code will return to the instruction following the coprocessor instruction, in this case the DSP multiply- accumulate instruction. This instruction will be repeated with identical operands, so the resultant Q flag state will be correct after the DSP multiply- accumulate instruction is executed. Under these conditions, the bug is benign, since to the main program code stream will maintain the proper Q flag and the emulation code does not rely on a specific value of the Q flag.

A single exception to this rule is the following code sequence, which contains a move from coprocessor instruction followed by a DSP multiply-accumulate instruction that has the source accumulate value Rn dependent on the coprocessor instruction:


```
MRC    p9, 0, r2, c0, c0, 0
SMLABB r7, r9, r0, r2
```

In this case, the SMLABB instruction will initially erroneously generate the Q flag based on the value of r2 prior to the MRC instruction. The emulation code will update r2 with the correct value, and return to the SMLABB instruction. However, the Q flag is sticky, and if the initial r2 prior to the MRC caused an overflow, and the value of r2 written by the MRC emulation code does not, then the Q flag will remain set, potentially causing a failure in the main program code.

Note This problem is not exhibited if the dependency is in either of the multiply operand source registers Rm or Rs. These are the second or third field of the SMLABB instruction shown above.

For the case of hardware debug watchpoints, the user should be aware that the state of the Q bit read out from the processor may be prematurely set if a watchpoint is hit on a load or store instruction that occurs directly before a DSP multiply-accumulate instruction. This can be determined by inspection of the code stream around the halted program counter address.

4.1.5 Workaround

The workaround is to place at least one instruction, like a NOP, between the coprocessor and the DSP multiply-accumulate instruction.

4.2 Breakpoint Instructions and Branch Prediction in Halt (Hardware) Debug Mode (r0p0, r0p1)

4.2.1 Summary

When debugging using halt (hardware) mode such as when attached to Multi-ICE™, one must be aware of issues with using the software breakpoint instruction BKPT when branch prediction is enabled. A breakpoint instruction placed at the speculatively issued target instruction of a predicted branch will trigger irrespective of whether the prediction is correct or not.

4.2.2 Description

The BKPT instruction is an ARM or Thumb mode instruction, with encoding of `E12xxx7x` in ARM mode and `BExx` in Thumb mode, where 'x' represents an immediate value. If branch prediction is enabled, and halt mode debugging has been enabled, a BKPT instruction inserted in the code stream as the first instruction following a predicted conditional branch will always halt the core. This occurs when the BKPT instruction is inserted:

1. At the address immediately following a forward conditional branch. This branch is predicted not taken, so the instruction immediately following the branch is speculatively issued.
2. At the target address of a backwards conditional branch. This branch is predicted taken, so the instruction pointed to as the target of the branch is speculatively issued.

BKPT instructions placed after unconditional, predictable branches or branch-and-link instructions do not trigger this problem. A BKPT instruction inserted as or after the 2nd instruction speculatively issued as the target of a predicted branch will not trigger this problem.

4.2.3 Conditions

This problem can only occur during use of halting (hardware) debug. It cannot occur in normal operation or when using monitor mode (non-stop) debug. It will occur only when a software BKPT instruction is placed at the target of a predicted conditional branch, as described above.

4.2.4 Implications

The failure mechanism associated with this errata is a disruption of the program flow. Assume the following code sequence is being debugged in halt mode:

```
        CMP   r0, #0
        BNE   Forward
        BKPT  0
        …
        …
Forward …
        …
```

This errata causes the ARM1020E to always halt at the breakpoint instruction irrespective of the value of *r0*. The debugger reads out the current program counter (*pc*) value to determine where the code execution was stopped, and the *pc* will point to the breakpoint instruction. Code resumption would occur at the address immediately following the breakpoint instruction, leading to the incorrect code stream being executed.

There are no implications for the use of hardware (CP14 register) breakpoints in any debug mode.

There are no implications for monitor mode (non-stop) debug or normal operational mode.

4.2.5 Workaround

The ideal solution would be to disallow BKPT instructions that exist as the target of a predicted branch. However, in practice, this is difficult to enforce, as backwards conditional branches are predicted taken, and a debugger usually cannot determine whether an instruction is a predicted target of a branch.

So two workable solutions exist:

1. When doing halt-mode debugging and utilizing BKPT instructions, branch prediction should be turned off.

2. The halt-mode debugger should only use hardware (register) breakpoints when debugging the ARM1020E target.

4.3 ARMV5TE SMULxy multiply instruction failure under special conditions (r0p0)

4.3.1 Summary

The ARMV5TE SMULxy multiply instruction may fail when followed by any multi-cycle multiply instruction.

4.3.2 Description

When any ARMV5TE SMULxy, the only single-cycle multiply instruction, is held over in Execute by a previous instruction AND the next instruction is a multi-cycle multiply or multiply-accumulate, the ARMV5TE SMULxy multiply will fail.

4.3.3 Conditions

All of these failures require that the first ARMV5TE SMULxy multiply be held in Execute by a preceding instruction. Holding the next instruction in the Execute stage can happen for many reasons; the example below illustrates one. The reason LDMIA holds the next instruction in the Execute stage is because r7 is also in the register list. Since it is possible for a data abort occur on the transfer, internal core hold signals prevent the overwriting of r7 until the transfer is successful. The side effect of this is to hold off the next instruction in the pipeline.

Example:

```
LDMIA    r7,{r3,r4,r7} ; Holds next instruction because of possible abort.
SMULBB   r10,r13,r13   ; Held in Ex by preceding instruction.
UMLAL    r1,r9,r2,r6   ; Multi-cycle multiply instruction.
```

Scenarios:

```
1)   SMULBB   ; Held in Ex by preceding instruction.
     SMLATT   ; Fails.     // ARMV5TE: 16 x 16 + 32

2)   SMULBT   ; Held in Ex by preceding instruction.
     SMULWT   ; Fails.     // ARMV5TE: 32 x 16

3)   SMULTB   ; Held in Ex by preceding instruction.
     SMLAWB   ; Fails.     // ARMV5TE: 32 x 16 + 32

4)   SMULTT   ; Held in Ex by preceding instruction.
     SMLALTT  ; Fails.     // ARMV5TE: 16 x 16 + 64

5)   SMULBT   ; Held in Ex by preceding instruction.
     SMULTT   ; Passes…    // ARMV5TE: 16 x 16 (single-cycle)

6)   SMULTB   ; Held in Ex by preceding instruction.
     MUL      ; Fails.     // ARMV4T:  32 x 32

7)   SMULTT   ; Held in Ex by preceding instruction.
     MLA      ; Fails.     // ARMV4T:  32 x 32 + 32

8)   SMULBB   ; Held in Ex by preceding instruction.
     UMULL    ; Fails.     // ARMV4T:  32 x 32

9)   SMULBB   ; Held in Ex by preceding instruction.
     UMLAL    ; Fails.     // ARMV4T:  32 x 32 + 64
```

Situations which will fail are:

- ARMV5TE SMULxy multiply followed by any ARMV4T multiply.
- ARMV5TE SMULxy multiply followed by any ARMV5TE multiply other than a SMULxy.

Situations which operate **correctly** are:

- Any ARMV4T or ARMV5TE multiply followed by ARMV5TE SMULxy multiply.
- Any ARMV4T multiply followed by any ARMV4T or ARMV5TE multiply.

4.3.4 Implications

If the conditions of this errata are met, the ARMV5TE SMULxy multiply instruction will produce the wrong result. Code containing ARMV5TE SMULxy multiply instructions with other multi-cycle multiply instructions immediately afterwards should be scrutinized.

Note that a very common sequence of instructions (particularly in DSP algorithms) is

```
LDMIA
SMULTT
SMLABB
SMLATT
.
.
.
```

where data and coefficients are multiplied and summed in an accumulator, usually within the context of a loop. Although adding an additional cycle with a NOP instruction will decrease the performance of the algorithm, the effect should be relatively small. Multi-cycle multiple such as SMLAxy are generally grouped together, allowing an issue rate of one multiply every two cycles. In a long sequence of multiply-add instructions, an extra cycle should not severely impact code performance.

4.3.5 Workaround

Insert a NOP instruction between the ARMV5TE SMULxy multiply instruction and any adjacent multi-cycle multiply instruction.

Example:

```
LDR      r6,[r8]        ; Holds next instruction of data cache miss.
SMULBT   r10,r12,r13    ; Held in Ex by preceding instruction.
AND      r0,r0,r0       ; NOP to hold off multi-cycle multiply.
UMLAL    r1,r2,r10,r12 ; Multi-cycle multiply instruction.
```


4.4 Incorrect halting (hardware) debug interaction with exceptions (r0p0, r0p1)

4.4.1 Summary

An interaction between certain exceptions and debug triggers can cause unexpected halting and entry into hardware debug state. In this event the debugger will see the core halt with MOE (method of entry) bits set according to the debug event but PC, R14, CPSR and SPSR set according to the exception. This may prevent the debugger from returning execution to the correct point in the program. This problem can only occur during use of halting (hardware) debug. It cannot occur in normal operation or when using monitor mode (non-stop) debug.

4.4.2 Description

The bug occurs when a load or store instruction that Data Aborts is closely followed by an event that could trigger entry into hardware debug state. Relevant evens that trigger entry into hardware debug state are: breakpoint comparator hit, breakpoint instruction, JTAG halt request and external debug request (EDBQRQ)

When an instruction that Data Aborts is ahead of a debug entry trigger in the pipeline the Data Abort exception should take priority over hardware debug state debug entry. This bug causes hardware debug state to be entered at the same time the Data Abort exception sequence starts. The bug also occurs when an Interrupt (IRQ or FIQ) occurs just before an event that would trigger entry into hardware debug state.

When the bug occurs R14, the SPSR and CPSR are updated (correctly) according to the exception, but the core also (incorrectly) enters hardware debug state and sets the MOE (method of entry) field for the debug state entry trigger condition.

Conditions identified:

- A load or Store instruction that Data Aborts followed by breakpoint comparator hit. In this case the following occurs:

  R14_abort_bank   <= Address of load or store instruction that aborted+8
  SPSR_abort_bank <= CPSR at the point the load or store instruction that aborted
  CPSR <= Abort mode
  PC <= Data Abort Vector+8
  MOE <= Breakpoint hit (incorrect)
  Normal execution halts and hardware debug state is entered (incorrect)

  For correct operation the core would not halt, the Abort handler should execute, this should return to re-execute the load or store (that should not abort this time), after this execution should continue until the debug trigger is reached again.

  During incorrect operation the debugger will see entry into hardware debug state with the MOE field set to breakpoint and the PC pointing to the Data Abort vector. In cases where the debugger returns to normal program execution by rerunning the instruction that halted it will jump to the instruction at the vector and the program will restart correctly with the abort handler (albeit after an unexpected entry into debug state). The abort handler will execute and return to the main code where the debug trigger (if it is still set) will cause debug state to be re-entered (as would be expected for correctly operation). In cases where the debugger returns by jumping to the instruction following the halted one the program will not restart correctly. In this case the instruction at the Data abort vector is skipped and the instruction at the reserved vector (0x000_0014) is then executed in error.

- A load or Store instruction that Data Aborts followed by breakpoint instruction results a similar set of events to those described above.

- A load or Store instruction that Data Aborts followed by a JTAG Halt or EDBGQ results a similar set of events to those described above. However, after these debug triggers, execution is almost certain to return to the instruction that halted so program execution will restart correctly. The halt may occur slightly earlier than might be expected but as JTAG Halt or EDBGQ are unlikely to be closely correlated to a particular instruction this is unlikely to be a problem.

- An Interrupt request (IRQ) of Fast Interrupt Request (FIQ) followed by breakpoint, JTAG Halt or EDBGQ. This results a similar set of events to those described above with IRQ or FIQ mode substituted for Abort and the IRQ or FIQ vector substituted for the Abort vector.

4.4.3 Conditions

This problem can only occur during use of halting (hardware) debug. It cannot occur in normal operation or when using monitor mode (non-stop) debug. It will occur for:

1) Data Aborted Load or Store instruction followed by a breakpoint comparator hit.
2) Data Aborted Load or Store instruction followed by a breakpoint instruction.
3) Data Aborted Load or Store instruction followed by a JTAG Halt
4) Data Aborted Load or Store instruction followed by an external debug request
5) Interrupted instruction (IRQ or FIQ) followed by a breakpoint comparator hit
6) Interrupted instruction (IRQ or FIQ) followed by a breakpoint instruction
7) Interrupted instruction (IRQ or FIQ) followed by a JTAG Halt
8) Interrupted instruction (IRQ or FIQ) followed by an external debug request

4.4.4 Implications

The most common result of this bug will be an unexpected early debug halt. In some cases (in particular those involving Breakpoints) the debugger may return to the wrong instruction on exiting hardware debug state. This may cause some confusing behaviour during the use of halting mode debug. There are no implications for normal program operation or monitor mode (non-stop) debug.

4.4.5 Workaround

In cases where the run time to the debug even is relatively short and debug trigger is not correlated to the exception the simplest workaround is to re-run the debug experiment. It is unlikely the combination of events required to exhibit this bug will re-occur. This workaround is most likely to work for conditions involving IRQ, JTAG Halt and external debug request as these are least likely to be correlated to a specific instruction.

In cases where the debug trigger is correlated to a specific instruction (eg. breakpoint instruction) that is also correlated to a Data Abort it is possible to work around the bug by moving the breakpoint to a nearby instruction, or even the aborted instruction, provided this still allows the desired debug task to be completed.

In many cases it should be also possible to simply return to normal execution from an unexpected or suspicious halt. For example when the MOE bits indicate JTAG Halt, external debug request and the (adjusted) PC points to the Abort, IRQ or FIQ vector.

For cases where the MOE bits indicate a breakpoint and the PC points to the Abort or IRQ vector (or any address for which a breakpoint has not been set) it should be possible to return to normal program execution by adjusting the PC so that the debugger returns to and re-executes the instruction upon which it halted.

4.5 Unable to enter debug state on EDBGRQ or JTAG HALT in a predicted branch-to-self loop (r0p0, r0p1)

4.5.1 Summary

In halt mode, the processor may not be able to enter debug state if EDBGRQ or JTAG HALT occur in a predicted branch-to-self loop.

4.5.2 Description

In halt mode, if branch prediction is enabled, the processor may not be able to halt and enter into debug state if an external EDBGRQ is asserted or a HALT instruction is scanned in through the JTAG interface while the processor is executing a branch-to-self loop.

The branch-to-self instruction must be predicted for the problem to occur. This will generally be true for branch-to-self scenarios where the branch instruction address is odd-word aligned, i.e. bit 2 of the address is 1.

Both ARM and Thumb conditional and unconditional branches to itself exhibit this problem.

```
Example 1)        loop  B           loop

Example 2)        loop  B<cond>     loop
```

Branch-and-link instructions, BLs and BLXs, do not exhibit this problem.

4.5.3 Conditions

This problem can only occur if branch prediction is enabled and the EDBGRQ or JTAG HALT occur in a predicted branch-to-self loop, as described above.

The problem will not occur if branch prediction is disabled or if another instruction, like a NOP, is in the loop.

4.5.4 Implications

An external debugger, such as Multi-ICE, may not be able to halt the processor in a branch-to-self loop if branch prediction is enabled.

4.5.5 Workaround

There are two workarounds that will allow the processor to recognize the EDBGRQ or JTAG HALT and enter into debug state.

The first workaround is to place at least one NOP instruction in the loop.

```
Workaround 1)     loop  NOP
                        B     loop
```

The second workaround is to replace the branch instruction with a SUB to PC.

```
Workaround 2)     loop  SUB   PC,PC,#8  //#8 for ARM, #4 for Thumb
```


5 CATEGORY 3 ERRATA

5.1 AHB Compliance Problem With HLOCK Signal (r0p0, r0p1)

5.1.1 Summary

The HLOCK signal may assert for a single cycle near SWPs under very specific circumstances.

5.1.2 Description

The setup to this problem is the following execution stream:

store (to a bufferable region of memory)

store (to a bufferable region of memory)

swap (to a non-cacheable region of memory)

The two stores that are inserted into the Write-buffer need to be drained before the swap can take place. Hence, there is a small latency from when the swap is presented to the Bus Interface Unit (BIU) and the actual locked transfer takes place on the AHB.

The following waveform shows the AHB signals to/from the arbiter. It is worth noting, that the compliance problem is solely related to the communication between the ARM1022E and the AHB arbiter. None of the address/control or data signals are involved.

```
             _____       _____       _____       _____       _____       _____
HCLK      __/     \_____/     \_____/     \_____/     \_____/     \_____/     \___
               ______________________________________________________________________
HBUSREQ   ___/
               ___________                                                ________
HLOCK     ___/    (1)     \______________________________________________/
                             ____________________________________________________
HGRANT    __________________/     (2) (3)          (4)                  (5)
```

(1) HLOCK is erroneously asserted along with HBUSREQ for one HCLK cycle due to the pending swap instruction.
(2) The AHB verifier complains, that HLOCK is removed before being granted via HGRANT!!!
(3) The first store out of the Write-buffer takes place.
(4) The second store out of the Write-buffer takes place.
(5) The swap takes place as one locked (atomic) transfer.

5.1.3 Conditions

Executing Swap instructions to non-cacheable memory regions when there are multiple buffered stores still pending in the write buffer.

5.1.4 Implications

There should be no functional impact as a result of this behaviour. The arbiter should constantly monitor HLOCK, such that bus master handover are only prevented from happening during locked transfers. As the blip on HLOCK is only one cycle, the arbiter should not be prevented from changing the grants if need be. Worst case in terms of performance would be a one cycle 'freeze' in terms of which master is driving the bus.

5.1.5 Workaround

None required.

5.2 AHB AMBA Rev 2.0 Compliance Problem With HTRANS (r0p0)

5.2.1 Summary

During burst of sequential writes on the AHB, it is possible, under very specific conditions, for the HTRANS to indicate a transition from IDLE to SEQ.

5.2.2 Description

The following burst of sequential writes is initiated by the ARM1022E DBIU:

```
HTRANS   | IDLE | NSEQ | IDLE | SEQ  |
         |   ___|_     |  ____|______|_
HGRANT__|__/    | \____|_/    |      |
```

This is an illegal burst sequence and is not in compliance with the AMBA Rev 2.0 specification. The burst is not compliant because HTRANS should always transition from IDLE to NSEQ, never IDLE to SEQ. However, the data and address information for this transfer is still correct.

5.2.3 Conditions

The Write Buffer (WB) is full and the BIU is requesting the bus. After some time, the DBIU gets granted and the first write goes out correctly, the NSEQ transfer. Coincidently with the first write going out, the DBIU loses the grant. The DBIU then responds correctly by indicating IDLE in the next HCLK cycle. One cycle after the grant was removed from the DBIU, the arbiter re-grants the DBIU the bus again. Due the original loss of grant, the BIU should now restart the write burst with a NSEQ transfer. However, the DBIU resumes the original burst with a SEQ transfer instead.

Note: This errata can only occur if the HGRANT was de-asserted for one cycle. If the HGRANT is deasserted for more than one cycle, the correct HTRANS transfer will be correctly observed, IDLE to NSEQ. Additionally, for systems that implement multi-layer AHB or even AHB Lite protocol, this errata will not occur. This is because the HGRANT will not be de-asserted for the ARM1022E, thus preventing the IDLE cycle from being inserted between the NSEQ and SEQ transfers.

5.2.4 Implications

This incorrect HTRANS behaviour impacts AHB peripherals using HTRANS control signals to decide how to handle incoming write data. Whether the incorrect HTRANS transition, IDLE to SEQ, will really cause a problem is hard to quantify since the 2 transfers, NSEQ and SEQ, are actually sequential. Any peripherals using the HTRANS signals will be using address information anyway because the burst type, HBURST, is indicating INCR and there exists a possibility the transfer might be crossing a cache line boundary. This errata would have been more severe if the transfers were actually non- sequential.

5.2.5 Workaround

A workaround is only needed for systems containing AHB peripherals using HTRANS bus to steer the write data for burst transfers. For these systems, software should set the processor to Fast Interrupt mode. One effect of the Fast Interrupt mode is that the Write Buffer size changes from 8 entries to 4. A side effect of shrinking the WB size happens to restrict a statemachine, internal to the design, from reaching a state which cause this errata.

Systems that implement multi-layer AHB or AHB Lite will not have this errata and do not need this workaround.

5.3 CP15 control register 1 LT bit set causes erroneous behaviour (r0p0, r0p1)

5.3.1 Summary

Whilst in hardware debug mode, if the LT bit is set, the core incorrectly changes execution state, Thumb to ARM, when an LDR/LDM PC is executed.

5.3.2 Description

When the CP15 control register 1 bit, LT or L4 bit, is set, the core is expected to ignore execution state information that is encoded on bit[0] during loads to the PC and doesn't. The setting of the CP15 LT bit should have suppressed the action of a possible execution state change. In the case of loads to the PC, while executing Thumb instructions in hardware mode, the execution state incorrectly transitions to ARM mode when exiting debug mode.

5.3.3 Conditions

The following actions will demonstrate the erroneous behaviour:

1) Set the CP15 LT bit to disable LDR PC from setting T bit.

2) Enable Global Debug and Hardware mode.

3) Switch into Thumb state.

4) Execute a Thumb BKPT instruction in Hardware mode with debugger.

   R15 saved should be PC+4

5) Within the Hardware debugger's BKPT handler, execute

```
STR PC,[Rn]
…
LDR PC,[Rn]
```

6) Exit debug state by doing a data processing operation with the PC, e.g.

```
SUB PC,PC,#n
```

Upon exit of debug mode, the processor should be in Thumb mode, however it is in ARM mode instead.

5.3.4 Implications

When debugging Thumb code using a hardware debug system, it is possible that processor will resume execution into ARM mode when exiting debug. The code now executed will be combined Thumb half-words instructions interpreted as 32-bit ARM instructions wherein the intended code sequence different than originally expected.

5.3.5 Workaround

Leave the CP15 LT bit left cleared, i.e. LDR/LDM to PC setting T-bit behaviour will NOT suppressed. Debugger code should then be modified to have bit[0] of the data for any loads to the PC match the expected execution mode processor upon exit from debug mode.

5.4 DFT: HRESPI[1], HRESPI[0], HRESTP[1], HRESPD[0] Input Ports Do Not Have Wrapper Cells (r0p0, r0p1)

5.4.1 Summary

HRESPI[1], HRESPI[0], HRESTP[1], HRESPD[0] input ports do not have wrapper cells. This prevents observability of any logic connected to these ports during tests occurring external to the core that use the ARM1020E wrapper.

5.4.2 Description

All functional inputs and outputs should have wrapper cells. HRESPI[1], HRESPI[0], HRESTP[1], HRESPD[0] input ports do not have wrapper cells. These cells are used to observe logic external to the core during external test mode. If the wrapper cells are not there and the wrapper is utilized during test, any logic connected to these ports cannot be observed and test coverage is affected.

5.4.3 Conditions

External scan test mode.

5.4.4 Implications

Test coverage loss will occur without these wrapper cells.

5.4.5 Workaround

Any logic external to the core connected to these inputs must be registered right before the affected ports in order to prevent test coverage loss.