LEON3FT RETT Restart Errata

Technical note 2020-09-22

Doc. No GRLIB-TN-0018

Issue 1.1

CHANGE RECORD

Issue	Date	Section / Page	Description
1.0	2020-01-31		First issue of document.
1.1	2020-09-22	4.4 5.1 2.3 3	Clarify that BCH protected register file is subject to errata. Clarification for accesses to %l1,%l2. Corrected which bits represents the FT field. Described which SW distributions implement the workaround.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Scope of the Document

This document describes an errata for the LEON3FT processor for a corner case requiring both a trap handler returning to a specific instruction, and a radiation induced upset in a specific location in the instruction cache memory, to occur close in time to each other. The likelihood of triggering the errata is application dependent, but is under typical application scenarios very low. The document also presents a software workaround for the issue.

1.2 Distribution

LEON3FT users are free to use the material in this document in their own documents and to redistribute this document. Please contact Cobham Gaisler for inquires on other distribution.

1.3 Contact

For questions on this document, please contact Cobham Gaisler support at support@gaisler.com. When requesting support, include the part name if the question is a specific device or the full GRLIB IP library package name if the question relates to a GRLIB IP library license.

2 AFFECTED PRODUCTS

2.1 General

This document applies to all versions of LEON3FT prior to build 4249, where either instruction cache parity protection is used, or where register file parity protection with restart is used. In practice this means all LEON3FT designs prior to build 4249 are affected.

Designs based on commercial LEON3 are not affected. No LEON4 designs (FT or commercial) are affected by the errata.

2.2 Cobham components

The following Cobham components are affected by the errata:
- UT699
- UT699E
- UT700
- GR712RC
- LEON3FT-RTAX

The following Cobham components are not affected by the errata
- GR716 – Not affected, does not use instruction restart
- GR740 – Not affected, LEON4 is not affected by errata

2.3 How to check if a custom design is affected

If you are licensing GRLIB for use in your own FPGA or ASIC design, you can check the following conditions in the design's VHDL source to see if the erratum applies to your system:

1. Check the GRLIB revision. This can be seen in the file name of the downloaded release package, in the directory name after unpacking the release, and in the file lib/grlib/stdlib/version.vhd in the release file tree (constant grlib_build).
2. Determine if your design is using LEON3 by examining the top level and seeing if the instantiated processor core is LEON3 (leon3x, leon3s).
3. Examine the instantiation to determine if cache memory fault tolerance (cft generic is 1) or register file protection with restart is implemented (iuft generic is 1 or 3) .

In case the VHDL sources are not available, the status of the design can be checked from software as follows:
1. Check the GRLIB revision. This can be read out from the Plug'n'play information on the devices, by default at the lower 16 bits of address 0xFFFFFFF0. The revision is also reported automatically when connecting with GRMON

2. Determine whether you are using a LEON3FT by examining the plug'n'play information. This is also reported by GRMON when connecting. If the processor is a LEON4 then it is not affected, otherwise proceed with step 3 below.
3. Check the cache control register, by reading out at ASI 2 address 0. If the FT field (bits 20:19) are non-zero then then cache memory fault tolerance is implemented and the device is affected by the errata.
4. Check the register protection control register, by reading out application specific register ASR16. If the IUFT field (bits 15:14) are 01 or 11, and the EIUFT field (bits 19:18) are 00, then register file protection with restart is implemented and the device is affected by the errata.

3 IMPACT

If the suggested workarounds are not implemented on an affected system, the errata can result in one instruction not being executed after the trap handler returns, which in turn leads to software malfunction.

The probability of occurrence is very low as several factors will mask the issue, as discussed further in section 4.5.

If one of the presented workarounds is implemented, the risks are eliminated. The workarounds must be implemented in the low-level trap handlers which are written in assembly language as part of operating system or runtime code. Compiled code needs not to be changed. Only customers that have written their own low level trap handlers need to update their assembly language code.

Software workarounds have been implemented in the following run-times / operating systems using the approach of workaround #1 described in section 5.1:

- BCC-1.0.52
- BCC-2.1.1
- RCC-1.2.25
- RCC 1.3-rc8
- VxWorks 6.7 2.0.23
- VxWorks 6.9 2.1.3
- VxWorks 7 SR0620 2.6

Please see the distribution specific documentation for more information. For example in Bare-C Cross-compiler (BCC) refer to sections "Recommended GCC options for LEON systems" and "Recommended Clang options for LEON systems" in the BCC User’s Manual, version 2.1.1 and later, for details.

The patch implementing the workaround in the RCC-1.3-rc8 environment is linked below. The reference is provided as an example implementation including source code updates and comments:
https://git.rtems.org/danielh/rcc.git/commit/?h=rcc-v1.3-rc8&id=97314d0961ee62a067304911034dc810a3412bec

The following older versions should not to be used since their workaround implementations does not fully cover all the possible sequences triggering the GRLIB-TN-0018:

- BCC-1.0.51
- RCC-1.2.24
- VxWorks 7 SR0620 2.5

4 DETAILED DESCRIPTION

4.1 Triggering sequence

This errata will occur when a trap handler returns into user code and all of the below are true:

1. The trap handler returns using a JMPL,RETT instruction pair
2. The target of the JMPL instruction used in step 1 is also a JMPL instruction.
3. The RETT instruction gets restarted due to a single-bit error that gets detected by the ECC check built into the LEON3FT.

Notes:

- Normally the sequences used in step 1 are jmpl %l1; rett %l2 to return to the trapping instruction, or jmpl %l2; rett %l2+4 to return to the instruction after the trapping instruction.
- The JMPL opcode is used in standard application code for function return (ret and retl is alias for "JMPL %i0/%o0, %g0") and also for indirect function calls ("call %reg" is alias for "JMPL %reg, %o7") but not normal function calls (CALL <constant>) or branches.

4.2 Example

A more concrete example of when the general case described above would occur would be:

1. An application runs with instruction cache and interrupts enabled
2. An external interrupt (for example a timer tick) occurs, and execution is transferred to the trap handler.
3. The interrupt trap handler runs and returns normally to the code. As a side effect, the instructions of the trap handler gets cached into the instruction cache.
4. The application code continues executing from the instruction where it trapped. At some point while running, an SEU occurs in the instruction cache memory that holds the RETT instruction of the interrupt trap handler.
5. The interrupt occurs again, and happens to be timed so that it is taken on the RET instruction at the end of a procedure in the application code (ending with a standard RET,RESTORE pair). The execution transfers to the interrupt trap handler.
6. The interrupt trap handler runs (this time out of instruction cache) and returns to the application code. The parity error on the RETT instruction will be detected and cause the RETT instruction to be automatically re-fetched from memory and restarted.
7. The trap handler continues executing from the RETT instruction where it trapped
8. The application continues executing from the point where it trapped. At this point the errata occurs.

4.3 Errata behaviour

When the errata occurs, the target of the RETT instruction (i.e. nPC, or the second instruction returned to from the trap handler) will be incorrectly annulled in the pipeline.

In the example sequence in section 4.2, this means that when returning to the RET,RESTORE pair in step 7, the RESTORE does not get executed. The application will then continue executing in the wrong register window, and as the register contents are wrong, this likely leads to an application malfunction shortly thereafter.

4.4 Causes of instruction restart

There are several possible causes of the RETT instruction restarting. All causes are related to radiation-induced upsets in memory.

Restart due to an instruction cache data parity error on the actual RETT instruction stored in cache memory is the most likely cause of the restart, and the focus of most of this errata.

The instruction could also be restarted due to a instruction cache tag error on the cache line of the RETT instruction. For this to trigger, the RETT has to be at the start of a new cacheline, as otherwise an earlier instruction will catch the tag error and restart.

If the LEON3FT is implemented with a register file protection scheme that uses correction with pipeline restart, the errata can also be triggered if the RETT instruction restarts due to a parity error on one of its source registers. Typically the instruction used is “rett %l2”, in which case a parity error on the %l2 register would cause the instruction to restart. This additional cause of restart only exists if the register file is protected using 4-bit parity with restart (iuft=1) or BCH with restart (iuft=3).

4.5 Likelihood of occurrence

There are several factors that combined determine the probability of the errata on a system where workarounds have not been implemented.

4.5.1 Likelihood of returning to a JMPL instruction

One of the conditions required to trigger the errata, is that the trap handler must be returning to a JMPL instruction. The possible scenarios where this could happen depend on how trap handling is used in the operating system (or runtime) used and has to be analyzed for the specific application.

In a typical bare-C application scenario, a trap handler return to a JMPL would only happen if an interrupt is timed so that it is delivered on a JMPL instruction. Since interrupt handlers typically return to the instruction where the interrupt was delivered to continue the execution, this will cause a return to a JMPL instruction.

For operating systems with multitasking, a task switching scenario could cause this as well where a task is switched out exactly when it is at a JMPL instruction, and then at a later stage switched back in. However this can be seen as an indirect variant of the previous case, since an interrupt occurring on a JMPL would be the reason why the application was switched out at that point.

For operating systems implementing demand-paging with the MMU, such as Linux, it is also possible to be returning to any instruction when the instruction_access_exception handler returns execution to a page that was just paged in. This would happen if the first instruction executed on the new page was a JMPL instruction.

For operating systems where software traps (TA instructions) are used to perform system calls, the operating system typically returns to the following instruction after the software trap. The following instruction could be a JMPL instruction, and in that case then the trap handler will always be returning to that instruction following the system call. Unlike the other cases, this would be an entirely deterministic cause that would either happen always or never for a specific system call site.

4.5.2 Likelihood of triggering a RETT restart

The trap handler has to be already present in the instruction cache when the radiation-induced upset occurs in order for there to be a instruction data parity error. If the cache line holding the RETT instruction gets evicted between executions due to other instructions being fetched in, or due to the application flushing the cache at some point between the traps, then it will be fetched in anew when the trap handler returns.

Note that a restart will only occur once after a parity error has been introduced, since the restart leads to a re-write of the cache line and this will put the cache line back into an error free state.

For instruction cache tag errors, these can only trigger a restart of the RETT instruction when the JMPL,RETT sequence crosses a cache line boudary and RETT is the first instruction on the cache line to be executed. Unlike the instruction cache data error case, the tag error may cause a restart even if the cache line is not present in cache.

In order for the register file parity error to trigger the restart, an upset must be induced into the register file in the time window between when the trap is taken (and the return address is written into the register file), and when the trap handler returns.

4.5.3 Functional masking

Even when the errata triggers, the instruction that gets annulled may not have any functional impact. For example, a leaf function might return using a RETL,NOP sequence and annulating the NOP due to the errata does not make any difference on the execution.

4.5.4 Probability calculation

The rate of occurrence can be calculated as:
$r_{errata} = r_{errata,idata} + r_{errata,itag} + r_{errata,regfile}$

where the three terms correspond to the three possible sources of RETT restart (instruction cache data error, instruction cache tag error, and register file error).

$r_{errata,idata} = r_{SEU,bit}$ x 36 x $n_{rett,1}$ x $p_{rett\_to\_jmpl}$
$r_{errata,itag} = r_{SEU,bit}$ x 36 x n_{iways} x $n_{rett,2}$ x $p_{rett\_to\_jmpl}$
$r_{errata,regfile} = r_{SEU,bit}$ x 36 x p_{intrap} x $p_{rett\_to\_jmpl}$

$r_{SEU,bit}$ is the upset rate of a single bit of SRAM

$n_{rett,1}$ is the number of RETT instructions that could return to a JMPL instruction that are present in the instruction cache at any given time, on average

$n_{rett,2}$ is the number of RETT instructions in the code that are at the start of a cache line and that could return to a JMPL instruction. The instructions do not necessarily have to be present in cache.

$p_{rett\_to\_jmpl}$ is the probability that the next trap that is susceptible to the errata will return to a JMPL instruction.

p_{intrap} is the probability at any given time that the processor is executing one of the trap handlers susceptible, or in other words the portion of the total time executed that the processor is executing one of the sensitive trap handlers.

N_{iways} is the number of instruction cache ways, and is 4 for existing components.

5 WORKAROUNDS

There are several possible workarounds for the issue, listed in separate sections below. The first workaround is recommended in most cases as it is simpler.

5.1 Workaround #1: Disable Icache during return sequence

The instruction cache can be enabled or disabled dynamically through the processor's cache control register. Due to the pipelined design of the processor, the write to this register has an effective delay of a few instructions delay as the write takes effect when it reaches the memory stage.

By disabling the instruction cache a number of instructions before the JMPL,RETT sequence and then re-enabling the instruction cache immediately before the sequence, one can guarantee that the RETT is not fetched from cache, which makes the errata not occur.

For processors implementing restart on register file parity error, readouts of l1 and l2 are performed shortly before the last sta instruction in order to catch the register file parity error before the JMPL, RETT sequence accesses to register file.

The code has a loop at the beginning to ensure the instruction cache is not currently flushing when the workaround is run, this loop can be removed if it is known by design that this can not the case. If the trap handler wishes to flush the cache when the trap returns, the workaround could be modified to set the FI bit (bit 21) of the cache control register at the same time as re-enabling the cache.

Example assembler code for implementing this fix at the end of the trap handler is shown below:

```
        mov %psr, %l0       ! save condition codes
1:      lda [%g0] 2, %l3    ! read cache control register
        srl %l3, 15, %l4    ! check bit 15 if flush in progress
        andcc %l4, 1, %g0
        bne 1b              ! loop back if flushing
        ! let following instruction go into the delay slot
        andn %l3, 3, %l4    ! mask out DCS field
        ! align cache line boundary at this point for optimum performance
        sta %l4, [%g0] 2    ! write to disable Icache
        mov %l0, %psr       ! delay + restore condition codes
        or %l1, %l1, %l1    ! delay + catch rf parity error on l1
        or %l2, %l2, %l2    ! delay + catch rf parity error on l2
        sta %l3, [%g0] 2    ! write to re-enable Icache after rett
        nop                 ! delay to ensure first insn after gets cached
        jmp %l1
        rett %l2
```

5.2 Workaround #2: Software emulation of JMPL target

This workaround is based on the trap handler examining the instruction that is being returned to, and if it is a JMPL instruction, emulating it in software, updating the PC,nPC accordingly and then returning to the following instruction instead. This is significantly more complex but can be slightly more efficient once the code is in cache, since only a single 32-bit read to get the instruction is needed.

Note that to fetch the instruction, an MMU/Cache bypass read is used, therefore the workaround as written will not work on a system with MMU address translation.

The code assumes the returned to address is in l1 and l2, just as after the trap handler is entered, and that the psr (condition codes) is in the state it was when entering the trap handler. The code starts by taking a back-up copy of the PSR to %l0, if the trap handler already has a copy of the original psr then this could be removed.

```
return_to_pc_fix:
        mov %psr, %l0
        ! backup used regs in trapped window
        mov %i7, %l5
        mov %i6, %l7
        ! load opcode
1:      lda [%l1] 0x1C, %l3
        ! check if jmpl
        sra %l3, 19, %l4
        and %l4, 0x183f, %l4 ! extract bits 31:30 and 24:19
        subcc %l4, 0xfffff038, %g0
        bne,a 9f
         mov %l0, %psr
        ! calculate target address
        !  get rs1 * 4
        srl %l3, 12, %l4
        and %l4, 0x7c, %l4
        ! check if using the register for temporary
        cmp %l4, (15 << 2) ! o7
        be,a 3f
         mov %l5, %i6
        cmp %l4, (14 << 2) ! o6
        be,a 3f
         mov %l7, %i6
        !  get registers by restoring and executing one instruction out of
        ! a table using DCTI couple
        set 0, %i6
        set regmov_table, %i7
        add %i7, %l4, %i7
        restore
        jmpl %o7, %o7
         jmpl %o7+8, %g0
        save
        ! check immediate bit
3:      srl %l3, 13, %l4
```

```
        andcc %l4, 1, %g0
        ! extract immediate field
        sll %l3, 19, %l4
        srl %l4, 19, %l4
        bne,a 2f
         add %i6, %l4, %i6
        ! imm=0, get rs2*4
        sll %l4, 2, %l4
        and %l4, 0x7c, %l4
        ! get rs2 register using same technique as for rs1 above
        cmp %l4, (15 << 2) ! o7
        be,a 2f
         add %l5, %i6, %i6
        cmp %l4, (14 << 2) ! o6
        be,a 2f
         add %l7, %i6, %i6
        set regmov_table, %i7
        add %i7, %l4, %i7
        restore
        jmpl %o7, %o7
         jmpl %o7+8, %g0
        save
        ! check for unaligned target address
2:      andcc %l7, 3, %g0
        bne 8f
         ! let srl below use delay slot
        ! swap over i6(next-nPC)->l2(nPC)->l1(PC)->o7
        mov %l1, %o7
        mov %l2, %l1
        mov %i6, %l2
        ! write back o7 to destination
        srl %l3, 23, %l4
        and %l4, 0x7c, %l4
        ! check if updating the registers used for temporaries
        cmp %l4, (15 << 2) ! o7
        be,a 1b
         mov %o7, %l5
        cmp %l4, (14 << 2) ! o6
        be,a 1b
         mov %o7, %l7
        set regmov_table2, %i6
        add %i6, %l4, %i6
        restore
        jmpl %o6, %o6
         jmpl %o6+8, %g0
        b 1b
         save

9:      mov %l5, %i7
        mov %l7, %i6
        nop
        jmpl %l1, %g0
rett2:  rett %l2
```

```
8:      ! jmpl to unaligned target, jump over to unaligned trap handler
        !  Note the tbr.ftt field will not be changed by the jump
        mov %l0, %psr
        mov %tbr, %l4
        andn %l4, 0xfff, %l4
        jmpl %l4+0x70, %g0
        nop

return_to_npc_fix2:
        mov %l2, %l1
        b return_to_pc_fix2
         add %l2, 4, %l2

regmov_table:
        add %o6, %r0, %o6
        add %o6, %r1, %o6
        add %o6, %r2, %o6
        add %o6, %r3, %o6
        add %o6, %r4, %o6
        add %o6, %r5, %o6
        add %o6, %r6, %o6
        add %o6, %r7, %o6
        add %o6, %r8, %o6
        add %o6, %r9, %o6
        add %o6, %r10, %o6
        add %o6, %r11, %o6
        add %o6, %r12, %o6
        add %o6, %r13, %o6
        unimp ! add %o6, %r14, %o6
        unimp ! add %o6, %r15, %o6
        add %o6, %r16, %o6
        add %o6, %r17, %o6
        add %o6, %r18, %o6
        add %o6, %r19, %o6
        add %o6, %r20, %o6
        add %o6, %r21, %o6
        add %o6, %r22, %o6
        add %o6, %r23, %o6
        add %o6, %r24, %o6
        add %o6, %r25, %o6
        add %o6, %r26, %o6
        add %o6, %r27, %o6
        add %o6, %r28, %o6
        add %o6, %r29, %o6
        add %o6, %r30, %o6
        add %o6, %r31, %o6
regmov_table2:
        mov %o7, %r0
        mov %o7, %r1
        mov %o7, %r2
        mov %o7, %r3
        mov %o7, %r4
        mov %o7, %r5
        mov %o7, %r6
        mov %o7, %r7
        mov %o7, %r8
        mov %o7, %r9
        mov %o7, %r10
        mov %o7, %r11
        mov %o7, %r12
        mov %o7, %r13
        unimp ! mov %o7, %r14
        unimp ! mov %o7, %r15
        mov %o7, %r16
        mov %o7, %r17
        mov %o7, %r18
        mov %o7, %r19
        mov %o7, %r20
        mov %o7, %r21
        mov %o7, %r22
        mov %o7, %r23
        mov %o7, %r24
        mov %o7, %r25
        mov %o7, %r26
        mov %o7, %r27
        mov %o7, %r28
        mov %o7, %r29
        mov %o7, %r30
        mov %o7, %r31
```

5.3 Workaround #3: Use MMU to prevent JMPL/RETT pair from being cached

For an operating system already using the MMU, a workaround could be to:

1. Modify all trap handlers that are susceptible to the errata, so that they branch to a shared location where there is a JMPL,RETT pair.
2. Ensure that the shared JMPL,RETT location is on a separate (4KiB) page in memory.
3. Modify the MMU page tables to make the page with the shared JMPL,RETT pair uncached.

The exact way of performing step 2 and 3 depends on the operating system design.

6 ROOT CAUSE DETAILS

6.1 Background on JMPL execution

In the SPARC architecture, dynamic jumps where the address is based on register values, are performed using the JMPL instruction. These are used for function return (using assembler aliases ret and retl) and for function pointer calls (using the assembler alias call %reg). The JMPL instruction, like most other control transfer instructions on SPARC, has delayed control transfer where the jump has a one instruction delay and the instruction right after the JMPL (often called "delay slot") is executed.

The target address of the JMPL instruction is not resolved until the execute stage in the pipeline, and therefore has a delay of 3 cycles before the target instruction is executed. Logic in the processor ensures only one instruction after the JMPL (the delay slot of the JMPL) is executed, and that any following instructions get annulled.

The execution of JMPL followed by regular single-cycle instruction is illustrated in the figure below. Other cases are possible, for example where the delay slot instruction is a multi-cycle instruction, but those details are not relevant for this errata.

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: JMPL→B						
1	A+4: NOP	A: JMPL→B					
2	(jmpl-annul)	A+4: NOP	A: JMPL→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: JMPL→B			
4	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: JMPL→B		
5	B+4: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: JMPL→B	
6	B+8: (…)	B+4: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: JMPL→B

It is possible (and legal) to put another JMPL instruction in the delay slot of a JMPL instruction. The effect of this is that only one instruction at the target of the first JMPL instruction is executed, then on the following instruction the second JMPL target "overrides" the destination and excution continues from there. This can be used in assembler code, for example as a way to execute a single instruction out of a table.

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: JMPL→B						
1	A+4: JMPL→C	A: JMPL→B					
2	(jmpl-annul)	A+4: JMPL→C	A: JMPL→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: JMPL→B			
4	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: JMPL→B		
5	C: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: JMPL→B	
6	C+4: (…)	C: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: JMPL→B

A special variant of the JMPL,JMPL pair is used at the end of trap handlers, using the instrruction sequence JMPL,RETT. The RETT instruction is very similar to JMPL but has some additional side effects needed when returning from a trap handler (see the SPARC specification for details). The execution flow in the pipeline is exactly the same as for JMPL:

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: JMPL→B						
1	A+4: RETT→C	A: JMPL→B					
2	(jmpl-annul)	A+4: RETT→C	A: JMPL→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B			
4	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B		
5	C: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B	
6	C+4: (…)	C: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B

Since SPARC requires the RETT instruction to always be in the delay slot of a JMPL instruction, the RETT instruction is normally never executed on its own. However, an instruction restart of the RETT instruction can cause the RETT to be re-run on its own, with the target of the preceding JMPL instruction in its delay slot, which may be any type of instruction.

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: RETT→B						
1	A+4: NOP	A: RETT→B					
2	(jmpl-annul)	A+4: NOP	A: RETT→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: RETT→B			
4	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: RETT→B		
5	B+4: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: RETT→B	
6	B+8: (…)	B+4: (…)	B: (…)	(jmpl-annul)	(jmpl-annul)	A+4: NOP	A: RETT→B

6.2 Failing case

The failing case occurs when RETT is executed with a JMPL in its delay slot. The annul logic does not correctly handle this case in the same way as it would for the other JMPL,JMPL and JMPL,RETT cases. The target of the RETT is annulled, as shown in red in the below table:

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: RETT→B						
1	A+4: JMPL→C	A: RETT→B					
2	(jmpl-annul)	A+4: JMPL→C	A: RETT→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: RETT→B			
4	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: RETT→B		
5	C: (…)	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: RETT→B	
6	C+4: (…)	C: (…)	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	A+4: JMPL→C	A: RETT→B

The full sequence where the errata triggers from a restarted RETT is shown below. The restarted RETT instruction is shown in yellow, the RETT and following instructions get annulled when the pipeline is flushed at cycle number 7.

Cycle	Fetch	Decode	Regfile	Execute	Memory	Exception	Writeback
0	A: JMPL→B						
1	A+4: RETT→C	A: JMPL→B					
2	(jmpl-annul)	A+4: RETT→C	A: JMPL→B				
3	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B			
4	B: JMPL→D	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B		
5	(irest-annul)	B: JMPL→D	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B	
6	(irest-annul)	(irest-annul)	B: JMPL→D	(jmpl-annul)	(jmpl-annul)	A+4: RETT→C	A: JMPL→B
7	-	-	-	-	-	-	-
8	A+4: RETT→C	-	-	-	-	-	-
9	B: JMPL→D	A+4: RETT→C	-	-	-	-	-
10	(jmpl-annul)	B: JMPL→D	A+4: RETT→C	-	-	-	-
11	(jmpl-annul)	(jmpl-annul)	B: JMPL→D	A+4: RETT→C	-	-	-
12	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	B: JMPL→D	A+4: RETT→C	-	-
13	C: (…)	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	B: JMPL→D	A+4: RETT→C	-
14	C+4: (…)	C: (…)	(jmpl-annul)	(jmpl-annul)	(jmpl-annul)	B: JMPL→D	A+4: RETT→C