MIPS R4000MC Errata, Processor Revision 2.2 and 3.0

May 10, 1994

MIPS Technologies Inc.
2011 N Shoreline Blvd
PO Box 7311
Mountain View, CA 94039-7311

Additional errata which affect uniprocessor designs and may affect multiprocessor configurations are listed in the MIPS R4000PC, R4000SC Errata, Silicon Revision 2.2 and 3.0.

1. A store instruction whose tags match a primary cache line, completes into the primary cache at the beginning of the certain stalls. Then, during the stall, an intervention comes in that invalidates or changes the tags of the store to shared. When the R4000 runs again, the store looks at its tags and signals a cache miss although its store has actually completed into primary cache. During the cache miss of the store, incorrect data is sent out under the update protocol.

Workaround: There is no workaround for this problem when using the update protocol. Invalidate protocol should be used in place of updates.

2. Replace this errata with the text in errata 5.

3. External requests may cause an incorrect state change in the secondary cache when the R4000 is in a data or instruction cache miss on a non-coherent page.

Workaround: Do not use the cacheable, noncoherent TLB page attribute in a multiprocessing system.

4. When the CU bit in the Config register is set to one, the R4000 should issue updates on the Store Conditional (sc) and Store Conditional Doubleword (scd) instructions instead of the protocol specified by the TLB page attribute. The R4000 ignores this bit and issues invalidates.

Workaround: Update protocol should not be used for the store conditional instructions. Previous errata prevent the use of the update protocol for TLB pages.

5. Store-load interlock breaks strong ordering. This situation can occur under the following conditions:

> 1. A store-load interlock is created when a load immediately following a store tries to access the same cache line. This load can be either to the same address or to an address which aliases to the same cache line location.
> 2. The store misses in the primary.
> 3. The stall sequence for the processor contains two stages.
> 4. First, the secondary is accessed to see if the line is present. After getting the data, which may not be the correct data because the tags do not match, the processor executes a restart sequence which is when the secondary tags are compared.
> 5. Second, there is a secondary cache miss after the restart, a stall for a cache state change or the processor must issue an invalidate or update request.

The problem arises if an intervention request is signaled to the processor between the two stages of the stall caused by the primary cache miss. To handle the store/load interlock correctly, the primary cache is written during the first stage of the stall and during the second stage of the stall.

This may create two problems. First, an intervention request that comes in after the first stage of the stall but before the second stage, will be satisfied before the store completes in the second stage of the stall. The intervention response will contain the new data. Later, the processor issues an update or invalidate which violates strong ordering.

Second, if the store was a store conditional, the intervention will receive the new data although the store conditional may never complete. In this case the intervening processor was returned incorrect data.

Workaround: Whenever a store to a sharable line is followed immediately by a load to a sharable line and the store and load have the same primary cache address, i.e. the 12 LSBs of the store address and load address match, insert a noop (or a non-memory instruction) between the store and load. A less fine-grained fix would be to just detect the store to sharable line followed by a load to sharable line without considering the address match and insert the noop or non-memory instruction.

New code sequence for general solution:

```
STORE instruction<SHARABLE line>
NOOP
LOAD  instruction<SHARABLE line>
```

6. With the system interface configured with ECC checking, the response to a Snoop request contains primary cache parity on the SysADC bus rather than ECC.

Workaround: Since a snoop request does not return data, the information present on the SysAD and SysADC busses should be ignored even if incorrect data is present.

7. If a store hit in the primary cache causes a state change stall and an external request invalidates the target of the store, the store will miss in and generate a read request for the target line. In this case, the check pins, SysADC, will not be driven during the read request.

Workaround: During a read request, ignore the SysADC pins.

8. Do not split the command and data cycles of an update cycle. Under certain conditions, the R4000 will prematurely take ownership of the system interface bus before the data cycle is issued to the R4000 if there are any cycles between the update command and data cycles.

Workaround: Do not split the command and data cycles of an update cycle.

9. Under the conditions listed below, the EB bit in the CacheErr register is incorrectly set.

1) A store targets a shared line in the primary cache
2) The tag in this line has a parity error
3) Under this condition, the processor will stall due to a data cache miss and the CacheErr register is set.
4) As the processor comes out of the data cache miss and before it vectors to the CacheErr exception vector, there is an instruction cache miss and a pending external request which targets the same line with the parity error.

Under these conditions, the EB bit will get set although there was no parity error. The EB bit implies that both a data and instruction parity error have occurred. In this case, there was only a data parity error.

Workaround: The EB bit is meaningful only if the ER bit indicates instruction error.

10. If a secondary cache line that is being replaced matches the address currently stored for a load linked, the cache line retained bit may not always be set in the read command issued for a cache refill..

Workaround: Software must guarantee that the load link address is not replaced by the instruction block that contains the load link instruction.

11. Processors might not function in “lock-step” properly because the timer in CP0 (Count Regis-

ter) may not synchronize across multiple processors at reset. As a result, the timers may increment on different clock edges causing the processors to fall out of lock step.

Workaround: Do not use the Count Register as a timer if more than one processor needs to function in lock-step with each other.

12. Under the following conditions, a load linked/store conditional loop may not function correctly:

1. An intervention request is issued to the processor which targets the same line as a store conditional
2. The store conditional stalls in the TC pipeline stage due to the intervention request.
3. The page in the TLB is marked with the Invalidate coherency attribute.

In this case, the R4000 responses with the updated store conditional data as if the store had completed successfully. When the R4000 resumes execution out of the stall, the store conditional fails; implying, the store was never completed. This causes the retry of the load linked/store conditional loop to load the data it had successfully stored.

Workaround: The success of the sc instruction should be redefined to be: 1) if ll bit =1 OR 2) if the ll bit is 0 AND the if store due to sc in the target location succeeded. The success of the store can be checked by reading the target location and comparing with the data planned to be stored. This works only if the data to be stored by sc is different from the data that existed in the target location. Here is a piece of code as an example:

```
/*****************************************************************************
*  Function llsc_loop uses the Load-Linked Store-Conditional sequence to
*  replace all or part of a 32 bit memory word.
*  If the Store-Conditional was not successful, delay before a retry of the
*  Load-Linked Store-Conditional sequence.
*  Call with the target address in a0 (register 4),
*                       mask in a1 (register 5),
*    and new data to be stored in a2 (register 6).
*  This function uses a3 (register 7).
*  This function returns the read value of the semaphore in V0 (register 2).
*****************************************************************************/

        .set    noreorder
LEAF(llsc_loop)
retry:
        ll      v0, (a0)        # Load Linked Reg 2 from (Reg 4)
        and     a3,a1,v0        # Extract the bytes not to be changed
        or      a3,a3,a2        # Merge & prepare the new data to be stored
        or      a2, a3, 0       # Save this new value to be stored by sc (should be different
                                # from the existing value at this address)
        sc      a3, (a0)        # Store Conditional Reg 7 to (Reg 4)
        bne     a3,0,ret        # If sc succeeded (a3=1), go to retrun
        nop                     # BD

        mfc0    a3, $9          # Load from C0_COUNT register
        nop
        nop
        andi    a3,a3,0x7       # Take last 3 bits to makeup a random-delay-count
```

```
loop: bne     a3,0,loop     # Semi-Random wait loop before retrying load-link
      subu    a3,a3,1       # Subtract loop count

      lw      a3, (a0)      # Read the data to check for the expected data.
                            # If sc fails but the store succeeds then assume
                            # that 'sc' is successful and return.
      beq     a3, a2, ret   # If Expected data = stored data, then return
       nop                  # BD
      j       retry         # Retry ll/sc again, as 'sc' failed
       nop                  # BD
ret:  j        ra
      nop
      .set    reorder
END(llsc_loop)
```

13. The load linked/store conditional link may not break if two processors try to access locations which map to different primary cache lines but the same secondary cache line.

Workaround: Have only one lock per secondary-block and align the lock address to a 32 word secondary-block size.

14. . Under the following conditions, a cache operation on primary cache may corrupt data in the secondary cache.

1. A primary cache cache-operation stalls because a write back is required.
2. An "Intervention request" is accepted by the processor to invalidate the same target line

The processor issues the correct data in response to the intervention request. But after the response is completed, the processor tries to complete the writeback of the invalidated data and will, incorrectly, set the SCAddr to 0 corrupting that secondary cache line.

Workaround: Do not use cache operations which involve a writeback on primary cache (Hit/Index_Writeback_Inv_D or Hit/Index_Writeback_D), except when the conditions listed above cannot occur. These cache operations can be synthesized by executing a dummy load (lw r0, (rx)) to an address, which would map to the same primary line but to different secondary line. This load would then writeback the dirty primary data to the correct secondary line.

Another solution is to use the corresponding cache-operations on the secondary cache instead of the primary cache, which would ensure correctness, at the expense of some performance hit.

15. Under the following condition, the DADDIU instruction can produce an incorrect result:
If this instruction generates a result value that would cause an overflow condition to occur (even though this instruction does not take an overflow exception) then the result value will be correct in bits 0-31 but bit 31 will be replicated through bits 32-63 (so it looks like a 32bit sign-extended value). The overflow condition is defined when the carries out of bits 62 and 63 differ (two's compliment overflow).

Workaround: There is no workaround for this problem.

16. Dirty shared mode may generate incorrect command sequences.
The problem occurs when the following sequence of events takes place:

1) store issued to a Dirty Shared or Shared line (line A)

2) processor invalidate is initiated and held up because of the deassertion of RdRdy*
3) an external request to the same or a different line is received (lineB).
4) RdRdy* is asserted.

At this point the processor should reissue the invalidate to line; however, it, incorrectly, issues one of the two requests:
1) in the Shared case, it issues coherent read to lineA
2) in the Dirty Shared case, it issues read with write forthcoming to lineA

The first case causes extra traffic but no serious problem; however, the second case, could be fatal since the processor issues the read for a line which it has modified and owns. The processor could end up with wrong data unless the read response uses the write data supplied by the processor.

Workaround: Do not use Dirty-Shared mode if RdRdy is used to control processor requests.

17. When Create-Dirty-Exclusive-SD cacheop is performed on a line which is present in the processor in Shared or Dirty Shared state; the processor invalidates the line before modifying it to Dirty Exclusive state. This might create the following problem: If there is a snoop or an intervention to this line, while the processor is waiting for IvdAck, the processor could send an incorrect response with an Invalid state instead of Shared or DShared state.

Workaround: There is no workaround for this problem.

18. External Updates to a line, which exists both in the PICache and PDCache at the same time, causes the copy of the line in the SCache and PDCache to be updated but does not change the state of the copy in the PICache. If the line is in the SCache and the PICache, only, then the processor properly updates the SCache line and invalidates the PICache line; and if the line is in the SCache and the PDCache, only, then the line gets updated in both secondary and primary caches, as expected.

Workaround: Do not allow a line to exist in both PDCache and PICache if an update protocol is used or use "write invalidate" protocol for the instruction space.

19. In this following sequence:

```
ddiv      (or ddivu or div or divu)
dsll32    (or dsrl32, dsra32)
```

if an MPT stall occurs, while the divide is slipping the cpu pipeline, then the following double shift would end up with an incorrect result.

Workaround: The compiler needs to avoid generating any sequence with divide followed by extended double shift.

20. The processor sends Read Request with incorrect value of the *"Link Address Retained"* bit. This error occurs when the following sequence of events takes place:

1) ICache miss to a line replacing link address in scache.
2) ICache Read request is stopped by de-asserting RdRdyB
3) An external request comes during this time and R4400 has to regenerate the address and command.

When the processor regenerates the Read Request after responding to the external request, it compares the link address register with a different address than the instruction address that caused the miss. As a result, sometimes it incorrectly sets or resets the *"Link Address Retained"* bit.

The consequence of incorrectly setting the *"Link Address Retained"* bit are not of any concern since the external agent would snoop assuming the line exist in shared state; but the processor would provide the state as Invalid. However, the consequence of incorrectly not indicating the *"Link Address Retained"* is significant since the atomic functionality could be broken.

Workaround: The hardware solution is to either not use the RdRdyB signal or in the case when the RdRdyB is used, to latch the retained bit when it occurs with the first Read Request even though the request is not accepted.