Errata for MIPS R10000 Microprocessor User's Manual, Version 1.1 (Jan' 95)

This errata supplements the "MIPS R10000 User's Manual, Version 1.1 released on on "www.mips.com" on Jan 24, 95" by Joe Heinrich

July 19, 1996

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

Page 10

In the Branch Unit subsection, fifth bullet:
Replace

"the Branch Link Quadword..."

To:

"the Branch Return Cache..."

Page 14

1) Replace the last sentence of the "Branch Prediction and Speculative Execution" section with the following:

Whenever the processor discovers a mispredicted branch, it aborts all speculative-executed instructions and restores the *processors state* to the state before the branch. However, the *cache state* is not restored (see the section on "*The Side Effects of Speculative Execution*").

2) At the end of the sub-section "Branch Prediction and Speculative Execution" add this paragraph:

The branch prediction can be controlled by the programming CP0 register 22, also called, *Diagnostic Register*. All branch likely instructions are always predicted as *taken*. This also means that the instruction in the delay slot of the branch likely instruction will always be speculatively executed. Moreover, the predictor is neither used nor updated by branch-likely instructions, so they do not affect the prediction of "normal" conditional branches.

Page 21

Add the following after the "Speculative Execution" sub-section:

Side Effects of Speculative Execution

In R10000 instructions are allowed to be fetched and executed speculatively. The decision was made to allow speculative requests on the system interface to improve performance. This is obviously one of the decisions that has a lot to do with the high performance of the processor. These side-effects, are harmless in a cached coherent operations; however, if unaware, could be hazardous in the system with non-coherent cached operations. These side-effects and the potential problems are explained below.

Speculatively fetched instructions and speculatively executed load or store to a cached address will initiate a *processor block read request* to the external interface if it misses in the cache; and may also cause change in cache state and/or data change which are not undone even if the speculation was incorrect and the instruction was aborted.

- **Speculative Processor Block Read Request to I/O address:**

Often accesses to I/O addresses cause side-effects. Typically, such I/O addresses are mapped to uncached region. Uncached Read/write occur as Double/single/partial-word read/write (non-block read/write) in R10000 and this uncached read/write are guaranteed to be non-speculative.

However, speculative block read requests to any "wild" physical address may occur if R10000 has some "garbage" value in a register, and it speculatively fetches data due to a *load* instruction specifying the register, or it speculatively fetches instructions due to a *jr* instruction specifying the register. Therefore, speculative block accesses to areas where to the load sensitive I/Os are mapped may cause problems.

- **Unexpected write-backs due to speculative *store* instruction:**

If a *store* instruction is speculated and then aborted, and if the target address of the speculative *store* instruction is missing in the cache, then the cache line is refilled and the state is marked to be *Dirty*; although, the refilled data might not be actually changed in the cache if the *store* instruction is later aborted. This could be a problem in certain cases such as the one shown below:

- The processor is storing some data sequentially to area "A", using a code-loop that includes store and cond.branch.

- DMA write operation is being performed to area "B".

- The DMA area "B" is contiguous to the area "A".

- The DMA operation is non-coherent.

- The processor does not cache any line of the DMA area "B".

If the processor and the DMA operations are performed in the sequence:

(1) Due to the speculative execution on the exiting of the code-loop, the next line to the end of the area "A", that is, the starting line of the area "B" is refilled to the cache, and the cache line is marked to be *dirty*.

(2) Non-coherent DMA write operation starts writing to the area "B".

(3) Due to a cache line replacement caused by later activities of the processor, the cache line of the top of the area "B" is written back to memory. Then, the first line of the DMA area "B" is overwritten by the old data, resulting an incorrect DMA operation.

The OS can restrict the writable pages for each user process and so can prevent a user process from accidentally/maliciously interfering with an active DMA space. The kernel, on the other hand, usually keeps a lot of xkphys or kseg0 addresses in the registers which, if speculatively used, offer no protection. User processes which have pages mapped to physical spaces which are not RAM may also have a problem. Obviously, the problem, exemplified above, can be avoided if the DMA is coherent.

- ### Speculative Instruction Fetch:

The change in the cache line state due to a speculative instruction fetch is not undone if the speculation was incorrect. This does not cause any program visible problems, except in some cases when the memory operations are non-coherent. Say the line is changed to *Clean Exclusive*. If this line also exist in some non-coherent space and later modified by some external component, the processor will have the older data.

Some recommended workaround for non-coherent cached systems

The suggestions listed below are not necessarily exhaustive. The solutions and the trade off are very system dependent. Any one or more of the items listed below might be suitable in a particular system, and only testing or simulations can confirm or verify.

a) External agent could reject the *processor block read requests* to the I/O location where speculative load could cause an undesired affect by simply returning an external *NACK completion response*.

b) Place a *serial* instructions like *cache barrier* or a *CP0 instruction* to prevent speculation beyond the point where speculative stores can occur. This could be at the start of *basic block* that includes instructions which can cause store with unsafe pointer. Note that *stores* to addresses like *stack-relative*, *global-pointer-relative* and pointers to non-I/O memory might be safe. Once the compiler prevents speculative stores, speculative loads could also be a potential hazard. Moreover to make sure there are no stale data into the cache due to any undesired speculative loads, parts of the cache referred by the address of the DMA read buffers could be flushed after every DMA transfer from the I/O devices.

c) Make reference to certain I/O space to be uncached by changing the cache coherency attribute in the TLB.

d) Generally, the arbitrary accesses to selected addresses can be controlled if the addresses are mapped via TLB. However, references to the *xkphys* region, which is unmapped, cached could cause hazardous affects to I/Os. One solution for *xkphys* is as follows:

Note that *xkphys* region is hard-wired into cached and uncached region; however, the cache attributes for the *kseg0* region can be programmed via the *config register*. First, turn off the *KX* bit but turn on the *SX* bit and *UX* bit in the *Status register*. This obviously disables *xkphys* region and restricts some of the 64bit address space but not all. In general, the system will either need coherent protocol or non-coherent but not both. This means that these cache attributes can be used by the external hardware to filter accesses to certain part of the *kseg0* region. Say, the cache attributes for the *kseg0* address space is selected via *config register* to be *cache-coherent* and the cache attributes in the TLB for the rest of the virtual space to be *cached-non-coherent* or *uncached*. Then the external hardware can be designed to reject all references to the memory in *cache coherent* mode, except to the pre-selected *safe* space within the *kseg0* in which there is no possibility of DMA transfer from I/O devices. Then, before the DMA read process and before the cached is flushed for the DMA read buffers, the cache attributes in the TLB for the address space to these buffers are changed from *non-coherent* to *uncached*. After the DMA read, the access modes are returned to the *cached-non-coherent* mode.

e) Use *conditional move* instruction with reverse condition in the speculated branch just before load/store instruction. The idea is to make all the assignments that are in the branch to be aborted be *safe*. This would work in some of the cases (in *basic block* with only one entry). Consider an example given below:

```
bne     r1,  r0, label
-----
-----
-----
-----
movn    r1,  ra, r0     # assign ra = 0 (safe value) if mispredicted.
ld      r4,  0(ra)      # Could create damaging read without movn.
-----
```

```
         -----
label:   -----
         -----
         -----
```

In the above example, without the *movn* the read to the address in register *ra* could be speculatively executed and later aborted. It is possible that this load could be premature and thus damaging. The *movn* guarantees that if there is a misprediction ra will have some address to which a read will not be damaging.

f) This is similar to the conditional-move fix, in that it protects speculation only for a single branch, but it seems that it might be cheaper than either the conditional-move or cache-barrier fixes, in some situations.

The fix uses the fact that branch-likely instructions are always predicted as taken by the R10000. Thus any incorrect speculation by the R10000 will always occur on the taken path, for branch-likely instructions. So, if we have code like the following:

```
        beql    r0, r1, label   #  if (p != NULL)
        nop
        sw      r2, 0x0(r1)     #     *p = val;
label:  -----
        -----
```

Then the store to (r1) will never be to an address of zero, because the store will never be executed speculatively. This fix is most useful when the branch is often taken, or there are few instructions other than memory operations that are being protected in the block. Also, similar to the conditional-move fix, this only protects fall-through blocks from speculation on the immediately preceding branch. Other mechanisms must be used to ensure that no other branches speculate into that block. However, if a block that dominates the fall-through block can be shown to be protected, this may be sufficient.

Page 35

In the description for the signal name: SCAAddr(18:0), change

"Duplicated 19-bit bus..." to "*SCBAddr* is complementary to *SCAAddr* 19-bit bus..."
(This change is an errata to the version 1.1 only, the version 1.0 was correct. In other words, it is reversing the errata for version 1.0)

Page 38

In the description for the signal name "SelDVCO":
Change:

"This signal must be tied to Vss."

To:

"This signal must be tied to Vcc."

Page 54

At the end of the page add this para:

The outputs and inputs at this interface are triggered by an internal SCClk. The relationship between the internal *SCClk* and the external *SCClk[5:0]/SCClk[5:0]** can be programmed during the boot time *SCClkTap* mode bits (see section 8.5 for detail on mode bits).

<u>Page 60 & 65</u>

At the end of the bullets in section 5.6 and 5.7 add the following paragraph:

The SCClk referred in secondary cache read or write timing diagram is an internal SCClk. The relationship between this internal SCClk and the *SCClk[5:0]/SCClk[5:0]** can be programmed by the *SCClkTap* mode bits (see section 8.5 for detail on mode bits).

<u>Page 165</u>

In the section "Data Array", 2nd sentence:
Change

"An even parity bit is used for rapid detection..."

To:

"An even parity for the 128 bits data is used for rapid detection..."

<u>Page 197</u>

In Table 12-1, Maximum Operating Conditions, change the

Parameter	**Symbol**	**Value**
Junction Temperature	T_j	20^o to 85^o C

To:

Parameter	**Symbol**	**Value**
Case Temperature	T_c	20^o to 85^o C

<u>Page 199</u>

At the end of the page, add the following:

All the JTAG output drivers are push-pull CMOS/TTL compatible, with Vcc (core) as the supply (They are independent of VccQ[SC, Sys]). All the JTAG inputs require full CMOS swings, as per the DC specifications shown in the table 12-3, "DC Characteristics for CMOS/TTL Configuration".

<u>Page 208</u>

1) Replace Table 13-2 with the following:

Table 13-2 R10000 599CLGA Thermal Characteristics - Preliminary

Parameter	Description	Value
T_C	Maximum case temperature	85° C
T_a *	Maximum ambient temperature	40° C

Parameter	Description	Value
P_{R10000}	Maximum power dissipation	30 watts
T_{ja}	Minimum temperature differential	45° C
Θ_{ca} *	Required case to ambient thermal resistance	1.5° C/W

Note: * Θ_{ca} is used as an example to calculate the ambient temperature T_c needed.

2) In the paragraph below Table 13-2, change the last sentence

"....case to ambient to increase thermal resistance, Θ_{ca} = 1.2C/W."

To:

"....case to ambient to increase thermal resistance, Θ_{ca}."

Page 245

Section on "Diagnostic Register", 2nd para:
Change

"The ten fields of the Diagnostic register..."

To:

"The eleven fields of the Diagnostic register..."

Page 248:

In Table 14-16, make the following changes[1]:

1) Remove the footnote to the "Instruction Issue" event in Table 14-16 (this was added in the errata for the version 1.0.

2) Change Event 6 for Counter 0: "Branches decoded" to "Branches resolved".

3) Change Event 14 for Counter 0: "Virtual coherency condition" to "Functional unit completion cycles".

4) Change Event 3 for Counter 0: "Stores issued" to "Stores (including store-conditional) issued"; and change Event 3 for Counter 1 "Store graduated" to "Stores (including store-conditional) graduated".

5) Change Event 9 for Counter 1: "Data cache misses" to "Secondary cache load/store & cache-ops operations".

Page 249:

Replace all the bullets under "The following rules apply:" with the contents below:

In describing the rules that are applied for the counting of each events listed in *Table 14-16*, following new terminology is used:

[1] The updated material reflect the functionality of silicon revision 3.0 and later revisions. For earlier silicon revisions, the absence of improvements that were made in silicon revision 3.0, are documented as a silicon errata.

"done" is defined as the point when the instruction is successfully executed by the functional unit but not yet *graduated*.

"Graduated" is defined as the point in time when the instruction is successfully executed (*done*) and it is the oldest instruction.

Secondary Cache Transaction Processing (SCTP) logic is on-chip logic where up to four internally generated and one externally generated secondary cache transactions to be processed are queued.

The following rules apply for the counting of each event listed in *Table 14-16:*

- **Event = 0 for Counter 0 & Counter 1:**

The counter is incremented on each PClk cycles.

- **Event = 1 for Counter 0:** ***Instructions issued.***

The configured counter is incremented on each cycle by the sum of the following a, b & c:

a) Integer operations marked as *done* on the cycle.

0, 1 or 2 such operations can be so marked on each cycle. Note that because these operations (except *mul/div*) are marked *done* on the cycle following their being issued to a functional unit, this is nearly identical to the number issued with the exception that re-issues are not counted.

b) Floating point operations marked *done* in the active list.

0, 1 or 2 are possible values. Note that because these operations take multiple cycles to complete it is possible for an instruction to be issued and then killed due to branch-mispredict or exception rollback before it is counted.

c) Load/store instructions first issued to the address calculation unit on the previous cycle.

Possible values 0 or 1. Note that *prefetch* instructions are counted as issued under this category. Also, the *load/store* instructions are counted as issued only once; even though they may be issued multiple times due to *DCache Tag* being busy or there are already four instruction or data cache misses in the *Secondary Cache Transaction Processing (SCTP)* logic waiting to be processed.

Moreover, when a Performance Counter is counting instructions issued, any instruction which does not go to the *load/store unit*, *integer functional unit*, or *FP functional unit* is not counted. This includes but is not limited to: *nops*, *bc1{f,t,fl,tl}, break, syscall, j, jal, jr, jalr, cp0 instructions*.

- **Event = 1 for Counter 1:** ***Instruction graduated*.**

The counter is incremented by the number of instructions that were *graduated* on the previous cycle. When an integer multiply or divide instruction *graduates*, it is counted as two *graduated* instructions.

- **Event = 2 for Counter 0:** ***Load/prefetch/sync/CacheOp issued.***

Load instruction is counted as issued only once; even though it may be issued multiple times due to *DCache Tag* being busy or there are already four *load/store* that had cache misses and are waiting in the *SCTP* logic.

- **Event = 2 for Counter 1:** ***Load/prefetch/sync/CacheOp graduated.***

  Up to four *loads* can *graduate* in one cycle.

- **Event = 3 for Counter 0:** ***Stores (including store-conditional) issued.***

  This counter is incremented if a *store* instruction was issued to the address-calculation unit on the previous cycle. Note that a *store* is only counted as issued only once; even though it may be issued multiple times due to DCache Tag being busy or there are already four load/store that had cache misses and are waiting in the *SCTP* logic

- **Event = 3 for Counter 1:** ***Stores (including store-conditional) graduated.***

  At most one *store* can *graduate* per cycle.

- **Event = 4 for Counter 0:** ***Store-conditional issued.***

  This counter is incremented if a *store-conditional* instruction was issued to the address-calculation unit on the previous cycle. Each *store-conditional* is counted as issued only once; even though it may be re-issued due to *DCache Tag* being busy or there are already four instruction or data cache misses in the *SCTP* logic waiting to be processed.

- **Event = 4 for Counter 1:** ***Store-conditional graduated.***

  At most one *store-conditional* can *graduate* per cycle. This counter is incremented on the cycle following the graduation of a *store-conditional* instruction.

- **Event = 5 for Counter 0:** ***Failed store conditional.***

  This counter is incremented when a *store-conditional* instruction fails.

- **Event = 5 for Counter 1: Floating-point instructions graduated.**

  This counter is incremented by the number of FP instructions which *graduated* on the previous cycle. Any instruction that sets the FP *Status* register bits (EVZOUI) is counted as a *graduated* floating point instruction. There can be 0 to 4 such instructions per cycle.

- **Event = 6 for Counter 0: Conditional branch "*resolved*".**

  This counter is incremented when a conditional branch is determined to have been predicted correctly or not. This is known as the branch being "*resolved*[1]." For floating-point conditional branches, multiple branches may be *resolved* in a single cycle, but this counter can only be incremented by 1. As a result, if multiple FP conditional branches are resolved in a single cycle, this count will be incorrect. This is normally a very rare event.

- **Event = 6 for Counter 1: Quadwords written back from primary data cache.**

  This counter is incremented once on each cycle that a quadword of data is written from primary data-cache to secondary cache.

- **Event = 7 for Counter 0: Quadwords written back from secondary cache.**

  This counter is incremented once on each cycle that the data for a quadword is written back from the secondary cache to the *outgoing buffer* located in the on-chip system-interface unit. Note that lines from the *outgoing buffer* might get invalidated by an *external request* and not get sent out of the processor.

---

1. In other words, this count is the sum of the conditional branches that are known to be mispredicted and those known to be correctly predicted.

- **Event = 7 for Counter 1: TLB refill exception (due to TLB miss).**

This counter is incremented on the cycle after the TLB miss handler is invoked. All TLB misses are counted, whether they occur in the native code or within the TLB handler.

- **Event = 8 for Counter 0: Correctable ECC errors on secondary cache data.**

This counter is incremented on the cycle following the correction of a single-bit error in a quadword read from the secondary cache data array.

- **Event = 8 for Counter 1: Branch mispredicted.**

This counter is incremented on the cycle after a branch is *restored* because it was mispredicted. Note that the misprediction is determined on the same cycle that conditional branch is resolved. The misprediction rate is the ratio of "*branch mispredicted"* count to the "*conditional branch resolve*" count.

- **Event = 9 for Counter 0: Primary Instruction cache misses.**

This counter is incremented one cycle after an instruction-refill request is sent to the *SCTP* logic.

- **Event = 9 for Counter 1: Secondary cache load/store & cache-ops operations.**

This counter is incremented one cycle after a request is entered into the *SCTP* logic, provided that the request was initially targeted at the primary data cache. Such requests fall into three categories:

1) Primary data cache misses.

2) Requests to change the state of secondary and primary data cache lines from clean to dirty due to stores that hit a clean line in the primary data cache.

3) Requests initiated by cache-op instructions.

- **Event = 10 for Counter 0: Secondary cache misses (instructions).**

This counter is incremented the cycle after the last quadword of a primary instruction cache line is written from the main memory, while the secondary cache refill continue.

- **Event = 10 for Counter 1: Secondary cache misses (data).**

This counter is incremented the cycle after the second quadword of a data cache line is written from the main memory, while the secondary cache refill continues.

- **Event = 11 for Counter 0: Secondary cache way mispredicted (instructions).**

This counter is incremented when the secondary cache control begins to retry an access to the secondary cache because it hit in the not-predicted way, provided the secondary cache access was initiated by the primary instruction cache.

- **Event = 11 for Counter 1: Secondary cache way mispredicted (data).**

This counter is incremented when the secondary cache control begins to retry an access to the secondary cache because it hit in the not-predicted way, provided the secondary cache access was initiated by the primary data cache.

- **Event = 12 for Counter 0: External *intervention requests*.**

This counter is incremented on the cycle after an *external intervention request* is entered into the *SCTP* logic.

- **Event = 12 for Counter 1: External *intervention requests* hits in secondary cache.**

  This counter is incremented on the cycle after an *external intervention request* is determined to have hit in the secondary cache.

- **Event = 13 for Counter 0: External *invalidate requests*.**

  This counter is incremented on the cycle after an *external invalidate request* is entered into the *SCTP* logic.

- **Event = 13 for Counter 1: External *invalidate requests* hits in secondary cache.**

  This counter is incremented on the cycle after an *external invalidate request* is determined to have hit in the secondary cache.

- **Event = 14 for Counter 0: Functional unit completion cycles.**

  This counter is incremented once on the cycle after at least one of the functional units - ALU1, ALU2, FPU1, or FPU2 - marks an instruction as "*done*".

- **Event = 14 for Counter 1: Stores, or prefetches with store hint to *Clean Exclusive* secondary cache blocks.**

  This counter is incremented on the cycle after a request, to change the *Clean Exclusive* state of the targeted secondary cache line to *Dirty Exclusive* state, is sent to the *SCTP* logic.

- **Event = 15 for Counter 0: Instruction graduated.**

  This counter is incremented by the number of instructions that were *graduated* on the previous cycle. When an integer multiply or divide instruction *graduates*, it is counted as two *graduated* instructions.

- **Event = 15 for Counter 1: Stores or prefetches with store hint to *Shared* secondary cache blocks.**

  This counter is incremented on the cycle after a request, to change the *Shared* state of the targeted secondary cache line to *Dirty Exclusive* state, is sent to the *SCTP* logic.

<u>Page 288</u>

For CVT.L.fmt, change the first sentence in the last paragraph:

> In the R10000, the conversion allows only up to 52 bits;...... The back conversion from a 52-bit long....."

To

> In the R10000, the conversion allows only up to 51 bits;...... The back conversion from a 51-bit long....."

<u>Page 321</u>

In the "*Virtual Coherency Exception* section", add the following at the end of the first paragraph:

> "When the hardware detects the *Virtual Coherency Exception*, it invalidates the lines in all the other segments of the primary cache that would cause the "aliasing" situation. It takes 6 extra cycles on the top of the cycles it takes to refill the line in the primary cache (the refill would have occurred even if there was no *Virtual Coherency Exception* detected).