
# Programming the MIPS64™ 20Kc™ Core

**Document Number: MD00332**
**Revision 0.01**
**August 1, 2003**

**MIPS Technologies, Inc**
**1225 Charleston Road**
**Mountain View, CA 94043-1353**

Copyright © 2003 MIPS Technologies, Inc. All rights reserved.

# Table of Contents

# Programming the MIPS64 20Kc Core

## 1. Introduction

This document describes where the MIPS 20Kc CPU core behaves - from the point of view of the most fussy, privileged software - in ways not specified by the MIPS64™ Architecture document [MIPS64][1]. Most of this variance is in the ‘‘coprocessor zero’’ or ‘‘privileged architecture’’ features, so most of the references in this document are to [MIPS64v3].

The document is arranged functionally. If you lose track of some CP0 registers, there’s a list in Appendix A with a cross-reference.

### Conventions

CP0 register numbers are denoted by `n.s`, where ‘‘`n`’’ is the register number (between 0-31) and ‘‘`s`’’ is the ‘‘select’’ field (0-7). If the select field is omitted, it’s zero. A select field of ‘‘`x`’’ denotes all eight potential select numbers.

## 2. A brief guide to the 20Kc hardware

20Kc is a high-end design and its implementation is more complex than MIPS Technologies’ other cores. It features:

- *2-way superscalar execution*: 20Kc can issue a pair of consecutive instructions in parallel. This won’t happen if one of the instructions depends on the result of the other, or if the instructions are not part of the same 4-instruction-aligned ‘‘fetch group’’, or for a host of other reasons. But you should be aware that the 20Kc can start and finish two instructions per clock.
- *Distributed pipeline control*: inside the 20Kc are a number of chunks of logic (‘‘functional units’’) which are involved in executing instructions. Rather than have some single overriding ‘‘conductor’’ who tells everyone what to do on every beat, each functional unit runs as and when it has inputs ready and somewhere to put the result.

  Why do you need to know that? So long as each unit runs as expected everything works smoothly: but it’s impractically difficult to predict the exact sequence of activity generated by a particular instruction sequence in a particular system. If you need to know exactly what happens with some code sequence, you should rely on measurement of a real or accurately simulated system.
- *Branch prediction and limited speculative execution* : the 20Kc’s relatively long pipeline means that several clocks elapse between fetching a branch instruction and the point where the CPU identifies the target of the branch (for a conditional branch the CPU must compute its branch condition, and for a jump-register must compute its target address.)

  The MIPS architecture requires that a single instruction following the branch (the ‘‘branch delay slot’’ instruction) will be executed regardless[2], but 20Kc could not compute a target address in time to keep going immediately after the branch target, as is possible with short-pipeline MIPS CPUs - and in any case it could be issuing two instructions per clock.

  Branches are common, and a 2-clock penalty on each branch would sap performance; so the 20Kc has circuits which guess the branch target as the branch instruction is decoded. Execution continues at the predicted address without missing a beat, but until the branch condition has been properly resolved these ‘‘speculative’’ instructions are not allowed to commit any state (they must not write any registers, or perform any write to memory).

  When the guess was wrong, the CPU must discard all speculative work and restart at the now-confirmed but different target, which is relatively costly (5 cycles in 20Kc). But it turns out that the guess can be right a

[1] References (in square brackets) are listed in §12.

[2] There’s a special case for ‘‘branch-likely’’ instructions; see the architecture manual, and note that such instructions are deprecated in code optimised for the 20Kc.

remarkably large proportion of the time; branch prediction is a big win for longer-pipeline CPUs.

20Kc has two branch predictors:

- For conditional branches there's a *branch history table* (256 entries for 20Kc). It's indexed by the low-order bits of the address of the branch instruction, and remembers the prior behaviour of any branch instructions with those particular low address bits: each entry is a 2-bit saturating counter incremented for branch-taken and decremented for not-taken.

  This really is as crude as it looks. No attempt is made to clear out the table when an entry is re-used with a new branch which happens to share the low address bits of a previous branch, for example. Nevertheless, it really does work.

  In 20Kc branch-likely instructions (**`beql`** etc) are *always predicted as taken*. This is done more for implementation convenience than efficicieny: these instructions are already deprecated by [MIPS64], so are provided only for compatibility. There's a penalty of five wasted pipeline clocks for any ''likely'' branch which is not taken.

- A *return address stack* is pushed by each **`jal`** instruction, and consulted on each **`jr ra`**, reasonably assumed to be a subroutine return[3]. A misprediction is inevitable when software returns through five or more levels of subroutine, but that's relatively rare.

Speculative instructions which are subsequently abandoned are not completely system- and software-invisible. They cause real fetches from the I-cache, though any cache miss causes speculation to be suspended. The speculative instruction stream interacts with the branch prediction mechanism, reads CP0 registers, and slows other activities by competing for resources.

Speculative instructions may not commit state: they will stall rather than write a value into the register file, store data into cache/memory, or perform any access on the external bus. In particular speculation is never permitted to cause an exception: if a speculative fetch or the partial execution of a speculative instruction encounters any condition which would cause an exception, everything waits until the original branch is resolved; you never see a ''speculative exception''.

Since 20Kc uses distributed control an instruction which is stalled does not stop everything: if it takes a while to resolve the branch condition a maximum of 18 speculative instructions[4] can be fetched to fill the queues in the various functional units.

- *"Replays" and unexpected delays* : The 20Kc's distributed control could cause a lot of delays, as control signals propagate between functional unit. To avoid this many of the control signals are defined to be available a clock too late for convenience: units continue optimistically based on reasonable expectations about their inputs and outputs.

  Sometimes something unusual happens, the reasonable expectations aren't met; then a functional unit overruns some output path, or runs out of inputs and starts computing junk. At this point the CPU consults a kind of ''journal'' of recent history, and can recover from overruns/underruns by winding back a few clocks and *replaying* the last few instructions.

  A replay causes an 8-cycle delay.

[3] Code generating conventions for all the popular MIPS ABIs insist that a return from subroutine is always done with a **`jr ra`**. even when (because the return address has been stored on the stack at some point) there is no particular reason to use that register.

[4] The precise number is an implementation detail and you should not rely on it; but this gives you some idea of how far speculation might reach.

# 3. Initialisation and identity

What happens when the CPU is first powered up? These functions are perhaps more often associated with a ROM monitor than an OS.

## 3.1. Config registers

The registers `Config` and `Config1` are mostly read-only 32-bit CP0 registers which contain information about the CPU's capabilities.

The few writable fields in `Config` are there for historic compatibility, and are typically written once soon after bootstrap and never changed again.

The MIPS64 standard permits up to four configuration registers: 20Kc has only these two. `Config1` is completely standard, but `Config` has some 20Kc-specific fields in the implementation-defined bits 30-16[5]. It also uses bit 3 to indicate a virtual I-cache - standard but worth noting.

Fields not shown are as required by the MIPS64 specification.

| *Name* | *Fields Bits* | *R/W?* | *Description* |
|---|---|---|---|
| **EC** | 30-28 | RO | External clock to pipeline clock multiplier:<br>`n` → multiplier is `(n+1)`.<br>Read-only, set by hardware. |
| **DD** | 27 | | Obsolete hardware setting, write zero |
| **LP** | 26 | RO | External bus width:<br>`0` → 64-bit<br>`1` → 32-bit<br>Read-only, set by hardware. |
| **SP** | 25 | RW | Parity options on CPU interface:<br>`0` → check data only<br>`1` → check commands too. |
| **TI** | 24 | RW | Controls connection of CPU counter/timer interrupt.<br>`0` → `Cause.IP7` from timer interrupt<br>`1` → `Cause.IP7` from CPU input signal, timer disabled.<br>In both cases `Cause.IP7` is really the logical OR of the condition above and the performance counter interrupt... |
| **TD** | 23 | | Writeable bits whose 1/0 value is reflected on a pin in the 20Kc on-chip interface, and are conventionally used to communicate test results.<br>Non-diagnostic users should write to zero. |
| **TF** | 22 | | |
| | 21-16 | | unused |
| | 15-4 | | as MIPS64 specification |
| **VI** | 3 | RO | Reads 1 to tell you the I-cache is virtual; this field is standard, though only added in Revision 1.0 of [MIPS64].<br>Read-only. |

*Table 3.1: Non-standard fields in the Config register*

[5] Many of these are writable, which is not quite in the spirit of the way [MIPS64] defines the `Config` registers.

### Post-reset value of Config.K0

This is not mandated by [MIPS64v3]. On 20Kc it is left set to ‘‘2’’ (uncached). Early system initialisation software will typically re-write it to ‘‘3’’ in order that *kseg0* will be cached, as expected.

### CPU information fields in Config1

Are set according to [MIPS64v3] to indicate:

- The TLB has 48 entries;
- Both the I-cache and D-cache are 4-way set associative with 32-byte lines and 256 sets/way (32Kbytes each).
- There is a performance counter.
- There is a watch register.
- MIPS16 is not available.
- There is an on-chip EJTAG debug unit.
- There is a floating point accelerator (FPU).

## 3.2. PRId register value

The PRId register is read-only, of course.

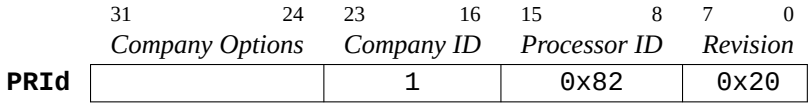


*Figure 3.1 PRId (processor ID) value*

- *Company Options*: comes from the SoC design (input pins to the core).
- *Company ID*: is ‘‘1’’ and represents MIPS Technologies Inc.
- *Processor ID*: of `0x82` is supposed to be changed if and only if there is some software-visible change in the specification. Note that a change which is fully accounted for in other registers - such as a differently-sized cache described in the `Config` registers - need not be recorded here.
- *Revision*: `0x20` in the first cores, but may increment with silicon revisions.

# 4. The memory map

The 20Kc implements a full 64-bit address map divided into all the standard segments shown in [MIPS64v3].

40 bits of virtual address space are available to user processes: so the user-mode virtual address region is 1Terabyte.

You should not need to hardcode the ‘‘40 bits’’ value. As [MIPS64v3] says:

> ‘‘Software may determine SEGBITS (the number of address bits required to span the user virtual address spaces) by writing all ones to the `EntryHi` register and reading the value back. Bits read as "1" from the `EntryHi.VPN2` field allow software to determine the boundary between the `EntryHi.VPN2` and `EntryHi.Fill` fields to calculate the value of SEGBITS.’’

20Kc generates a 36-bit physical address (a 64Gbyte addressible region). Again, see [MIPS64v3] for how to work this out from the TLB registers in a somewhat future-proof way.

# 5. Reads, writes and synchronisation

20Kc is a high-performance design and read operations are split into an address phase and a later data phase, with other bus operations happening in between.

## 5.1. Sequencing of loads and stores

- *Non-blocking loads*: the CPU supports four pending loads - either uncached loads or resulting from cache misses. Only a fifth load will cause it to stall.

  If any instruction references a register which is the subject of a pending load, everything stops until the data arrives.
- *Hit-under miss*: the D-cache continues to supply data on a hit, even though there is one or more cache refills pending.
- *Write-under-miss*: the CPU pipeline continues and can generate stores even though a read is pending. 20Kc’s interface is non-blocking too (reads consist of separate address and data phases, and writes are permitted between them), so this behaviour can often be visible to the system.
- *Miss under miss*: 20Kc can continue to run until it accumulates up to four pending read operations.
- *Ordering*: the address phase of uncached reads, cache refills and uncached stores are presented in order on the system bus. Cache writebacks (caused by data being pushed from the D-cache to make space for a line required to service a cache miss) are typically deferred until pending read traffic has been dealt with.

  Reads are never allowed to overtake writebacks to the same memory address.

### Uncached accelerated writes

20Kc permits memory regions to be marked as ‘‘uncached accelerated’’. This type of region is useful to hardware which is ‘‘write only’’ - perhaps video frame buffers, or some other hardware stream.

Such regions are uncached and not partial-word writable: but sequential word and doubleword stores in such regions (which span a whole cache-line sized and aligned memory block) are gathered into cacheline chunks each of which is written with a single burst cycle on the CPU interface. This burst is marked by per-word ‘‘valid’’ bits, so that unwritten memory words are not corrupted.

The burst write is normally performed when software writes to the last location in the memory block; but it can also be triggered by a **`sync`** instruction, or by a byte write[6] in the uncached accelerated space (which is otherwise a no-op: remember, partial word writes don’t work in an uncached accelerated region).

[6] Why both? It’s because the **`sync`** instruction is privileged, and sometimes the software pushing out the uncached data might be a user-level task.

## 5.2. Bus error/Cache Error

20Kc checks its incoming data bus (and the command buses too, if so configured) for parity. A parity error always results in a ‘‘cache error’’ exception, and it’s imprecise: that is, `ErrorEPC` does not generally point to the instruction which caused the error. Parity errors are barely recoverable, and should usually be processed as a controlled crash (which is often preferable to the uncontrolled behaviour which would follow if the CPU was fed with bad data).

### CacheErr register

This is a read-only register and is valid only after a cache error exception. Unused fields read zero.

| *Name* | *Bits* | *Description* |
|---|---|---|
| **ER** | 31-30 | Where was the error detected?<br>00 → Instruction Cache<br>01 → Prefetch Buffer†<br>10 → Data Cache<br>11 → Fill Store Buffer† |
| **ED** | 29 | set if the data field was wrong |
| **ET** | 28 | set if a cache tag field was wrong |
| **ES** | 27 | set if the error happened during an external request such as a cache invalidation. |
| **EE** | 26 | set by a parity error on the CPU interface |
| **EB** | 25 | set on an ‘‘instruction error’’† |
| | 24 | Reserved |
| **EW** | 23 | set when there was a tag error on an external request |
| | 22-15 | |
| **WA** | 14-13 | The cache way at which the error was detected. |
| **IN** | 12-5 | The cache index at which the error was detected. |
| | 4-0 | |

*Table 5.1: Fields in the CacheErr register*

### ErrCtl registers

In fact there are two of these, one for each cache: `DErrCtl` and `IErrCtl`. They are 32-bit registers of which the low 8 bits are used to access the data parity bits of the caches. When being read, the bits are valid following an ‘‘index load tag’’ instruction.

`DErrCtl` can also be written (setting the parity bits on a ‘‘store tag’’ instruction), but `IErrCtl` is read only. To store the bits you have to set `SR.CE` and then perform any store operation which hits in the cache. It seems unlikely you’d do this for any purpose other than diagnostics, and you’d need to refer to the full manual.

### Bus error exception

20Kc’s read protocol permits something elsewhere in the system to signal that data is bad (it’s a ‘‘bad’’ bit in the bus command encoding); it results in a a bus error exception. Typically this reports a failure of some subsystem to respond to the supplied address. The bus error is also imprecise, since the (non-blocking) load which caused it may have happened a long time ago.

If software knows that a particular read might encounter a bus error - typically it’s some kind of probe - it should be careful to stall and wait for the load value immediately, by reading the value into a register, and make sure it can handle a bus error at that point.

† The author is hazy as to what these things are; they are internal components of the CPU design which are not visible to software. It probably doesn’t really matter. Cache errors are fatal, and these fields are there to help you debug your logic. It’s time to pull out the full CPU manual and involve the hardware engineers...

# 6. Basic privileged operation, exceptions and interrupts

## 6.1. The counter/timer

20Kc's timer register `Count` increments at the pipeline clock rate (on many MIPS CPUs it increments at one half of the pipeline clock rate).

If you want counter/timer interrupts you must enable them to be delivered to `Cause.IP7`; see Table 3.1 above.

## 6.2. Status register

- *Implementation-defined bits*: `SR[17](CE)` is a writable bit used for cache diagnostics, see §5.2 above.

  `SR[16](DE)` is a writable bit set to disable cache parity error processing.

- *Reduced power mode*: set `SR.RP` to reduce the system clock rate by setting the system clock/pipeline clock multiplier to 1. It takes effect only when the CPU is not in exception mode (that is, when `SR.EXL` and `SR.ERL` are both zero).

  Note that in reduced power mode the counter register `Count` slows down, too. You might not have expected that.

# 7. Caches

Using [MIPS64v3], you can find out the size and shape of the I-cache and D-cache by reading `Config1`. 20Kc's caches are unusual; it has a virtually-tagged I-cache and a physically-indexed D-cache. As a result of that asymmetry it uses separate cache-maintenance registers for the I- and D-cache: they're `ITagHi`/`ITagLo`, `DTagHi`/`DTagLo`.

There are cache data registers too.

## 7.1. I-cache

20Kc's I-cache uses virtual addresses for both its index and tags; a feature which was not used in MIPS architecture CPUs until recently, and may require new support in some OS.

The advantage of a virtual cache is speed: you don't have to wait for the memory management hardware (the TLB) to translate the address before looking it up in the cache[7]. That means more MHz or a shorter pipeline, and both are very desirable. Naive virtual caches have two serious disadvantages, and it's easier to appreciate the 20Kc design if you appreciate what those might be:

- The same virtual address may be in use by many different Linux processes. Long ago when caches were small and CPUs were slow, you could just discard the cache contents when you did a process switch which changed the address map; but you can't reasonably do that with a 32Kbyte cache.

  So modern implementations like 20Kc extend the cache tag with a *address space ID* (associated with the particular address space). 20Kc uses the 8-bit ASID field already maintained in the `EntryHi` register for the same sort of purpose in the TLB. Note that when the OS needs to recycle an ASID - the 8-bit field gives you only 256 unique values - it is still necessary to discard the whole I-cache contents.

- Several different virtual addresses may map to the same physical memory - common (for various reasons) in Linux and Windows CE. So a virtual cache may store the same memory location more than once.

  For an I-cache that's not much of a problem; the CPU can't write to the I-cache, and the OS already has to do special things when it writes instructions. For a D-cache this would be a disaster, which is why nobody builds virtual D-caches any more.

There's a complication: the MIPS architecture defines a trick which makes some virtual address regions ''global'' - the hardware maps them to the same physical addresses regardless of the ASID. This is used widely used in Unix systems where mapped kernel regions are accessible to any process which is running a kernel routine. MIPS CPUs implement this with the ''`G`'' bit in a TLB entry; such an entry will match a virtual address regardless of the current ASID value. 20Kc's virtual I-cache must also have a ''`G`'' bit, and must do the same; a cache line with the ''`G`'' bit set allows a hit from a matching virtual address, regardless of the ASID.

### I-cache tags register

This register's layout is always machine-dependent; 20Kc's is particularly unusual because of the needs of the virtual I-cache. Note that information about the tag register fields should never be used by an OS - which at most can write zero values into the tag registers as part of cache initialisation - and is required only when debugging or for CPU/cache diagnostics.

The tag field is wide, so it helps to see the registers as a pair in Figure 7.1:

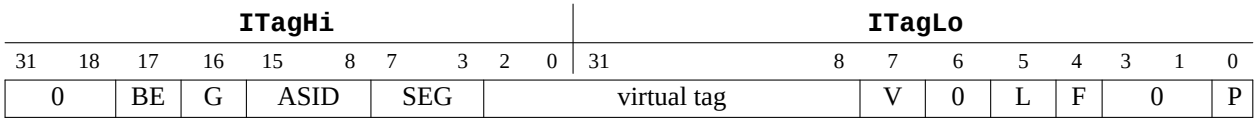


*Figure 7.1 I-cache tags registers ITagHi/ITagLo*

Where the fields are:

[7] Many other MIPS architecture CPUs - including the 4K and 5K core families - use a virtual cache index, but a physical address for the tag. But you still have to translate the address in time to check the tag, and that's more of a problem in the faster 20Kc.

- *BE* : why do we need an endianness bit here? Recall that 20Kc implements the `SR.RE` bit, which allows the CPU to switch endianness in user mode, so (with enough software support) you could run a little-endian application on a big-endian OS, and vice-versa[8]. Cached instructions can therefore survive across the endianness-change boundary.

  A MIPS instruction stream is a sequence of 32-bit objects, and has no inherent endianness. But 20Kc handles instructions in pairs on its 64-bit buses; so it needs to know its endianness to decide which of the two instructions presented on the 64-bit bus is first in sequence. Having the ''BE'' bit stored with the cache line ensures there is no timing race where instructions for a little-endian task might be unpacked big-endian.

- *G, ASID* : the ASID field and its ''global'' bit matching the data in this line, as discussed above. For translated addresses `G` will have been taken from the TLB entry; for untranslated addresses it's always set (untranslated address regions are available to all address spaces, thus ''shared'').

- *SEG* : the highest bits 63-62 of a MIPS 64-bit virtual addresses determine the ''segment'' of the address map in which this address lives, and thus how it's treated. In the XKPHYS segment (only) bits 61-59 are used to subdivide that segment into multiple sections mapping all physical memory, but with different cacheability attributes (see Table 7.1 on page 15 for encodings).

  Since in 20Kc none of the segments is more than $2^{40}$ bits in size, so addresses 58-40 may never be meaningfully other than zero; the cache doesn't have to track them.

- *virtual tag* : virtual address bits 39-13. Virtual address bits 12 and downward form the cache index, so don't need to be kept in the tag.

- *V* : valid bit. The big manual calls this `PState` for consistency with the D-cache, where there is more than one bit of line state.

- *L* : locked bit. See §7.3.

- *F, P* : cache housekeeping bits - one used to select cache lines for replacement, the other parity. Neither have any software significance.

## Programming 20Kc's virtual I-cache

With carefully designed OS support, a virtual I-cache need pose no great problems. But if you're only familiar with physically-tagged caches and how they work in systems, you need to change your own mental models quite a bit to adapt to a virtual cache.

So a virtual cache is a cached copy of previously-referenced instruction lines from some virtual address space.

An OS may maintain vast numbers of virtual address spaces, but the CPU's ASID is only 8 bits; no more than 256 address spaces can be represented by either TLB or I-cache entries at any one time. A process keeps its ASID either until the process itself is being terminated, or until the OS (having run out of valid new ASIDs) selects the process as the best target to be de-ASID-ised. A process without an ASID can have no entries in the TLB, no code in the I-cache, and is therefore far from runnable: the OS needs to keep track of this so it can restore the process if and when required. So:

- *When the OS takes back an ASID from a process* : it must discard all translations for that ASID from the TLB[9], and all lines using that ASID from the I-cache. The hardware offers little help in selecting entries for just one ASID, so you can expect the OS to discard all TLB entries and the whole I-cache contents.

When a process springs into existence with an ASID, it will have no matching TLB or I-cache entries. The TLB entries will appear as a result of TLB miss processing in the normal way, and won't be discussed further here.

I-cache entries will be generated as it runs. Since these cache the virtual space, these will cease to be valid if and only if the OS adjusts the executable contents of the address space, and that happens (for Unix/Linux) in three contexts:

---

[8] Note that (to the author's knowledge) this has not been used by any OS.

[9] If you're converting an OS to support a virtual I-cache for the first time, it is a useful guideline that where you need to remove entries from the TLB, you will usually also need to remove entries from the I-cache.

- *On an exec()* : the `exec()` function maps a new program image to a Unix process. This creates a complete new memory map, and therefore any previous I-cache entries become invalid. At this point all I-cache entries for this ASID must be discarded.
- *On an mmap()* : `mmap()` maps a file to some portion of a process memory space. By far it's most common use is as the mechanism underlying dynamically-loaded libraries. If this is the first time any valid contents have been associated with this part of the address space, there should be nothing to invalidate; otherwise you need to invalidate entries for this ASID and matching the range of addresses affected - and that range is likely to be sufficiently large that it is easier to invalidate the entire I-cache.
- *Whenever you write instructions* : if the CPU writes machine instructions for execution to anywhere. Linux/Unix don't do this for normal program loading: but the auto-generation of small pieces of code on a process' stack (''trampolines'') is a standard part of implementing signals. And interpreters can do it.

  In this case you need to first force a writeback of any D-cache copies of the locations affected, and then an I-cache invalidate to remove any stale copy of the data. This should certainly be done using an operation which only invalidates copies of the affected locations.

CPUs with physically tagged I-caches require the last of these, but not the first two.

You should note that a virtual I-cache *does not* usually need any attention when reading instructions from disk for a page-in from an application binary or shared library. These pages are not application-writable; you're either reading that data for the first time in the process' life (when there can be no I-cache copy), or you're just re-reading the same data as might have been visible at this virtual address earlier (in which case any long-lived I-cache entry is in fact valid). The OS will have hooks to provide necessary invalidation for physically-tagged caches in the disk drivers, and it may be difficult to establish that the context is a page-in...

One last unexpected feature: there is no connection between the algorithms by which the I-cache and TLB discard old entries to make room for new ones. So it's quite possible for instructions to be validly cached in the I-cache, even though there's currently no translation entry for the address in the TLB. You'd have to try hard to notice that, of course.

## Executing from "uncached" region

20Kc implements uncached execution as an I-cache miss; kind of obvious. Because it doesn't consult the TLB when looking up the (virtual) I-cache, the difference between cacheable and uncacheable regions is noticed after the cache miss, when no attempt is made to fill a new cache line.

This sounds the same... but it means that:

1. If you change the cacheability of a piece of code which is already in the I-cache, the CPU will still use it from the cache - unless you go in and software-invalidate it.

   Bear in mind that one way you can change cacheability is to use the `Config.K0` to make ''kseg0'' uncached - sometimes done for debug or in bootstrap code - and in this case too instructions which are already cached will continue to act cached until invalidated.
2. At reset time the cache is in an ''undefined'' state (which may contain fossil state from before the last reset). *Any* code you execute from anywhere except the permanently uncached ''kseg1'' region could potentially hit in the cache, returning garbage. So it's important to invalidate the I-cache early in the bootstrap process, even if you plan to run code uncached.

## 7.2. D-cache

20Kc's D-cache is both physically-indexed and physically tagged. This means that it's free of the ''cache alias'' problem which afflicts many MIPS CPUs.

### D-cache tags register

As for the I-cache tags, the information here should never be used by an OS - which at most needs to write zero values into the tag registers as part of cache initialisation - and is required only when debugging or for CPU/cache diagnostics.

The (nominal) `DTagHi` register doesn't exist: there is enough room for all required tag, line state and housekeeping bits in the 32-bit `DTagLo`, shown in Figure 7.2.

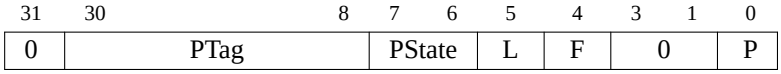


*Figure 7.2 D-cache tags register DTagLo*

The fields are fairly well-understood:

- *PTag* : physical tag - to be matched against bits 35-13 of the translated physical address.
- *PState* : a cache line can be in one of three states:

| *Code* | *State* |
|---|---|
| 0 | Invalid |
| 1 | Invalid |
| 2 | Clean Exclusive |
| 3 | Dirty |

  The ''valid but not written'' state is called ''clean exclusive'' to emphasise that 20Kc's simple coherency model does not allow two caches being kept coherent to share access to the same line of memory: to be made valid in one cache, any copy in any other cache must first be invalidated.

- *L* : set one to lock the line: see §7.3.
- *F, P* : housekeeping bits for the hardware: one used to keep track which line in a way was least recently filled, the other is a parity bit.

### D-cache DMA coherency

20Kc provides a mechanism which DMA transfer can use to maintain coherency. If that's enabled in your system, you will never have to invalidate or writeback the data cache because of DMA activity[10]

DMA coherency is a system option, and many 20Kc systems may not provide the interface: in that case you have all the usual problems afflicting software-managed writeback caches, which are commonplace on MIPS64 CPUs[11].

[10] Note that you can't run an OS with no explicit cache maintenance code on 20Kc. At a minimum, you still have to do a D-cache writeback when writing instructions - there's no automatic updating of the I-cache from the D-cache - and the I-cache must be handled explicitly, too.

[11] That is, you still have to invalidate a buffer region when an I/O device is supplying DMA data, and write-back the buffer region before an I/O device takes DMA data. And without hardware coherency, you have to be careful about segregating buffer data into separate cache lines...

### Cacheability options

Whether data can be cached, and how writes are managed, is influenced by a cache control code either obtained from the TLB or (for untranslated ‘‘kseg0’’ accesses) found in the `Config.K0` register field. The available values for 20Kc are shown in Table 7.1.

| *Code* | *Behaviour* |
|---|---|
| 0 | Cacheable, Noncoherent, Write Through |
| 2 | Uncached |
| 3 | Cacheable, Noncoherent (Writeback) |
| 4 | Cacheable, Coherent (Writeback) |
| 7 | Uncached Accelerated |

*Table 7.1: Page/Region read/write/cacheability codes*

Codes 2 and 3 are standard and required by [MIPS64v3]. The attributes are:

- *Cacheable* : this data can be read through the cache.
- *coherent/noncoherent* : influences code which marks a block read. An OS can mark pages which it knows can’t be affected by DMA at run-time as ‘‘non-coherent’’ and a system controller[12] would be freed from the burden of ‘‘snooping’’ such cycles.
- *writeback* : the normal way to handle CPU writes in the D-cache. In this kind of region stores happen only to the cache, not (yet) to memory. Data is only written to memory when the cache line is recycled to service a later miss (or explicitly written back with a **`cache`** instruction).
- *write through* : an optional setting, which causes all writes to go directly to memory. Write data is captured in the cache if and only if the location is already present in the cache.

  At high speeds this is amazingly inefficient, but it can be helpful when driving some memory-mapped device such as a video frame buffer.
- *accelerated* : a peculiar way of doing writes appropriate to some high-speed streaming output devices: see §5.1 above.

## 7.3. Cache locking

In both the I- and the D-cache software can set a line into locked state using the **`cache FetchAndLock`** instruction; it’s remembered by the the ‘‘locked’’ bit visible through the cache tag registers `ITagLo.L` and `DTagLo.L`.

See [MIPS64V2] for programming information.

A locked line can be invalidated either by a **`cache`** instruction or an external invalidation request. If you rely on cache locking, you need to take control.

Perhaps it’s worth saying here that cache locking is a feature which can be locally useful but globally pernicious; think hard before using it in a large and complicated system.

## 7.4. CacheOps

20Kc implements a full set of cache operations[13] including the (MIPS64-optional) ‘‘Fill_I’’.

20Kc also provides a pair of pairs of cache data registers `IDataLo`/`IDataHi` and `DDataLo`/`DDataHi` which return the D-cache line data contents during a read-tags cache operation. For diagnostics only.

[12] Note that the ‘‘Bonito’’ controller available on 20Kc ‘‘Malta’’ evaluation boards from MIPS Technologies Inc does not take advantage of this optimisation, so the coherent and noncoherent spaces behave identically in this case.

[13] A bug in some revisions of 20Kc makes the ‘‘hit invalidate’’ cacheop unreliable. The (fairly painless) workaround is to always prefer ‘‘hit writeback and invalidate’’.

# 8. TLB and translation

20Kc has a conventional MIPS64 TLB with 48 paired entries.

- *Care with duplicated entries during initialisation* : Software should be careful about initialising the TLB, though. In the fully-associative TLB it makes no sense to have two entries which could match the same virtual address; in some logic implementations an attempt to translate the same address through two entries could even damage the hardware.

  While all known MIPS-architecture CPUs provide some defence against such a translation, the 20Kc does something more; it takes a ‘‘machine check’’ exception if you try to set up such a dual translation.

  This exception is taken even when the translations both lead to entries flagged as ‘‘invalid’’, and even when the virtual address used in the entry is itself ‘‘untranslateable’’ - ie in KSEG0 or some other fixed-mapping region.

  While it’s fairly straightforward for an initialisation routine to fit around these requirements, it can be problematic when an OS is initialising a TLB which has already been initialised by a boot ROM; unless the ROM and the OS use identical algorithms, you’re quite likely to hit on the same addresses but in different slots.

  The solution is that as you form each address for your ‘‘invalid’’ TLB entries, you should first check that this address doesn’t match any existing entry (using a **`tlbp`** instruction). If it does, you should advance the address to the next invalid page address in your sequence, and try again until the probe shows no match.

- *Page sizes* : `PageMask` has encodings to support all MIPS64 page sizes from 4Kbytes to 16Mbytes (in ×4 intervals).

- *Non-standard cacheability attributes* : `EntryLo0`/`EntryLo1` registers have an extended ‘‘cacheability’’ code as described in Table 7.1 above.

- *How does* `Random` *count?*

  Counts down continually from the maximum value (47), wrapping back to 47 when it is found equal to the `Wired` register (it’s also set back to 47 if `Wired` is ever written).

  But there’s an extra trick: the hardware remembers the index of the last TLB entry used for translation, and avoids using that index in the write-random **`tlbwr`** instruction (it decrements the `Random` pointer twice in that case). This provides some last-ditch defence against unlucky sequences which create ‘‘beats’’ and cause a flood of TLB misses.

# 9. Floating point (CP1)

20Kc provides a standard MIPS64 FPA, but also provides the MIPS 3D ASE, described in [MIPS3D]. This includes ‘‘paired-single’’ operations (SIMD instructions which process both of a pair of single-precision values held in one 64-bit register).

It seems to have just one unexpected feature:

## Unimplemented exception used in integer/float conversions

20Kc’s hardware will only convert a limited range of values between integer and floating point - a limitation which is commonplace in MIPS architecture CPUs. Conversion of a quantity outside these ranges will result in an ‘‘unimplemented’’ exception and must be handled by software.

However, 20Kc’s limits are somewhat more restrictive than some other 64-bit MIPS floating point units, so an application new to 20Kc may experience more ‘‘unimplemented’’ exceptions in these cases

- *Single-precision floating point* : the hardware performs only conversions where the input value *N* is in the range:

  $-2^{23} \le N < 2^{23}$

  The FP format can in fact represent larger values precisely (to $2^{24}$); but the conversion algorithm uses up a bit.

  Many other 64-bit MIPS architecture FPAs will convert any value in the range of a 32-bit signed integer.

- *Double-precision floating point*: the hardware will perform those conversions[14] where the integer value *N* satisfies:

  $-2^{51} \le N \le 2^{51}-1$

  This limit is typical for conversions from integer to floating point, but some other 64-bit MIPS architecture FPAs will cope with larger numbers when converting from double-precision floating point to integer.

It may be clearer to set these out in a table.

| Integer → float: **`cvt.[ds].[wl]`** | | | | |
|---|---|---|---|---|
| Conv | *Most negative* | | *Most Positive* | |
| | *Input* | *Output* | *Input* | *Output* |
| **`cvt.s.w`** | $-2^{23}$<br>FF80.0000 | - E=128+22<br>M=0 | $2^{23}-1$<br>007F.FFFF | + E=128+21<br>M=7F.FFFE |
| **`cvt.s.l`** | $-2^{23}$<br>FFFF.FFFF.FF80.0000 | CB00.0000 | $2^{23}-1$<br>0000.0000.007F.FFFF | 4AFF.FFFE |
| **`cvt.d.w`** | no limit, all 32-bit values can be converted to double | | | |
| **`cvt.d.l`** | $-2^{51}$<br>FFF8.0000.0000.0000 | - E=1024+50<br>M=0<br>C320.0000.0000.0000 | $2^{51}-1$<br>0007.FFFF.FFFF.FFFF | + E=1024+49<br>M=F.FFFF.FFFF.FFFC<br>431F.FFFF.FFFF.FFFC |
| Float → integer: **`cvt.[wl].[ds]`**, **`round`**, **`ceil`**, **`floor`**, **`trunc`**. | | | | |
| Conv | *Most negative* | | *Most Positive* | |
| | *Input* | *Output* | *Input* | *Output* |
| **`cvt.w.s`**<br>**`cvt.l.s`** | - E=128+21<br>M=7F.FFFF<br>CAFF.FFFF | $-2^{23}$+0.5†<br>FF80.0000 | + E=128+21<br>M=7F.FFFF<br>4AFF.FFFF | $2^{23}$-0.5†<br>007F.FFFF |
| **`cvt.w.d`** | - E=1024+28<br>M=F.FFFF.FFFF.FFFF<br>C1CF.FFFF.FFFF.FFFF | $-2^{30}$-e‡<br>C000.0000 | + E=1024+28<br>M=F.FFFF.FFFF.FFFF<br>41CF.FFFF.FFFF.FFFF | $2^{30}$-e‡<br>4000.0000 |
| **`cvt.l.d`** | - E=1024+50<br>M=F.FFFF.FFFF.FFFF<br>C32F.FFFF.FFFF.FFFF | $-2^{52}$+0.5†<br>FFF0.0000.0000.0001 | + E=1024+50<br>M=F.FFFF.FFFF.FFFF<br>432F.FFFF.FFFF.FFFF | $2^{52}$-0.5†<br>000F.FFFF.FFFF.FFFF |

*Table 9.1: Limits of hardware conversion between integer and floating point*

## Notes on Table 9.1

- *Instruction conventions*: remember that **`cvt.s.w`** should be read as ‘‘convert from word to single’’, that the integer formats are **`w`** (‘‘word’’) and **`l`** (‘‘long’’), the floating point formats are **`s`** (‘‘single’’) and **`d`** (‘‘double’’).
- *Data representation*: each full value (and mantissa field values) are shown in hexadecimal. For floating point values we also show the sign field (‘‘+’’ or ‘‘-’’); the exponent value ‘‘E=xx’’, shown as $x + y$, where $x$ is the mathematical exponent value and $y$ is the ‘‘bias’’; and the mantissa ‘‘M=xx’’. [SEEMIPSRUN] describes the IEEE754 floating point formats.
- †: extreme floating-point values are not necessarily exact integers, so the result depends on the rounding rule. The integer values shown are the result of rounding to the nearest integer, with round-to-zero as a tie-breaker for values which are exactly *xx.5*.
- ‡: e represents a tiny quantity, one least-significant bit of double-precision format.

[14] Well, apparently it’s not so simple; Table 9.1 aims to be precise. Apparently FP double→64-bit integer conversions work for any exactly-representable quantity, but FP double→32-bit conversions are limited by a bit...

Don’t assume this is exactly right in this version.

# 10. Debug and visibility facilities

The 20Kc provides EJTAG debug, compliant with v2.6. You can set up to four instruction breakpoints and two data ‘‘breakpoints’’ (more often called ‘‘watchpoints’’). See [EJTAG].

There is a single watchpoint register provided outside of the EJTAG unit for compatibility with non-EJTAG CPUs, and a single performance counter register.

## 10.1. Watchpoint

There’s a single 64-bit watchpoint register usable for instruction and data, controlled through the `WatchLo`/`WatchHi` register pair, as shown in Figure 10.1. Software can tell there is only one watchpoint; the presence of further registers would be indicated by a 1 in `WatchHi[31]`.

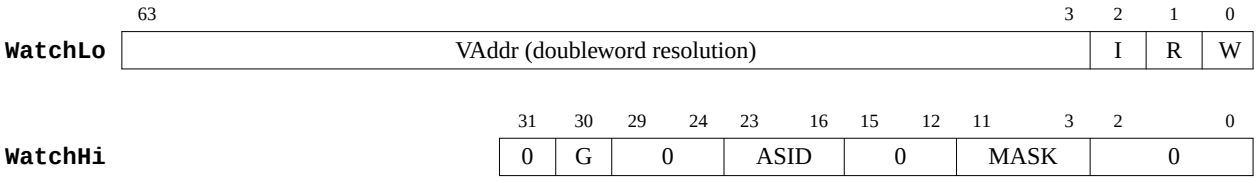


*Figure 10.1 Fields in the watchpoint registers WatchLo/WatchHi*

### About Watchpoint register fields

- *VAddr* : a 64-bit virtual address. Since it ends at bit 3 it only has a doubleword (8 byte) resolution, and will match on any access within the doubleword. But see `WatchHi.MASK` below for how you can coarsen the match further.
- *I, R, W* : trigger on any of Instruction access, data Read (load) or Write (store). Any combination is permitted; you can disable the watchpoint function by writing zero to all.

  [MIPS64v3] requires these to be set zero on reset to make sure you can’t get unwanted watchpoint registers before software can initialise `WatchHi`. But in 20Kc they are not so cleared. In theory this could cause an infinite loop, if you were unlucky enough to power up with a watchpoint matching an instruction in the very early post-reset initialisation code; in practice this is vanishingly improbable, but it is important to clear these bits out as early as possible after the CPU starts up.
- *G, ASID* : address-space ID and control. For user addresses you’ll usually set `WatchHi.ASID` so only addresses from the address space of the task under debug match. If the `WatchHi.G` (global) bit is set, `WatchHi.ASID` is ignored and only the virtual address is matched. If you set the address to one of the ‘‘unmapped’’ regions, you certainly want to set `WatchHi.G`.
- *MASK* : allows you to coarsen the resolution of the watchpoint to catch more addresses (up to a 4096 byte region). Each bit set in `MASK` eliminates the corresponding bit of the virtual address from comparision.

## 10.2. Performance counter

There is a single 32-bit performance counter with its control word. The 20Kc's performance counters are orientated more to the need of the implementors than the software developer: consult the list of events below.

The performance counter supplies an interrupt, active if the high-order bit of the counter is set. The control register bits are shown in Figure 10.2.

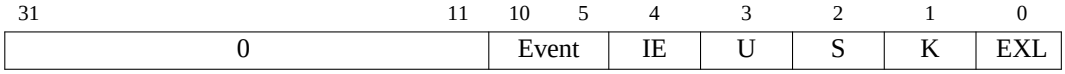


*Figure 10.2 Performance counter control register fields*

Where:

- *0* : reads zero, please write as zero. The MIPS64 specification requires bit 31 to be clear to indicate there are no further performance counters.
- *Event* : event code currently being counted. Supported values:

| *Value* | *What is counted* |
|---|---|
| 0x00 | CPU cycles |
| 0x01 | dispatched/issued instructions |
| 0x02 | fetch groups entering CPU execution pipes (a fetch group is an aligned group of up to four instructions) |
| 0x03 | computational FP instructions executed (doesn't count branches, loads, or **`mtc1`**/**`mfc1`** moves). |
| 0x04 | TLB refill exceptions |
| 0x05 | branch mispredictions |
| 0x06 | all branches |
| 0x07 | taken Joint-TLB exceptions |
| 0x08 | Where an instruction consuming load data is issued in the clock cycle immediately after its load, and the load misses in the primary cache, 20Kc uses a replay (whose overhead is usually lost in the cache refill time). This counts those events. |
| 0x09 | instruction requests from the IFU to the BIU |
| 0x0A | FPU exceptions taken |
| 0x0B | counts the total number of replays due to any of (a) requests from the load/store unit (b) load/use replays (as in `0x08` above) or (c) FPU exception prediction |
| 0x0C | **`jr ra`** (return) instructions that mispredicted using the return prediction stack. |
| 0x0D | all **`JR`** instructions executed |
| 0x0E | load/store unit requested replays |
| 0x0F | instruction that completed execution (with or without exception) |

- *IE* : set 1 to enable interrupt.
- *U, S, K* : set 1 to enable counting in any of user, supervisor or kernel mode as required.
- *EXL* : set 1 to enable counting when in an exception handler, too (ie when `SR.EXL` is set but `SR.ERL` is not).

# 11. Software-managed hazards

[This section currently reproduces the 20Kc manual section]

## 11.1. CP0 Hazards

Because resources controlled via Coprocessor 0 affect the operation of various pipeline stages of a MIPS64 processor, manipulation of these resources may produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a CP0 hazard exists.

The 20Kc processor implements hardware interlocks to resolve many CP0 instruction hazards: Table 11.1 shows those hazards which are not handled by the interlocks. The hazard is resolved by either a delay of a number of dispatch cycles (number of **ssnop**s required) or by the execution of some ‘‘barrier’’ instruction.

Changes to `EntryHi.ASID` and `Status` are not guaranteed to affect the instruction fetches until an **eret** instruction is executed.

| *Producer* | | *Consumer* | *Hazard On* | *until* |
|---|---|---|---|---|
| **LL** | → | **MFC0** | `LLAddr` | +4 cycles |
| deferred watch exception | → | **MFC0** | `Cause.WP` | +2 cycles |
| `EntryHi.ASID` | → | instruction fetches | EntryHi.ASID | **eret** |
| `Status` | → | instruction fetches | `Status` | **eret** |
| Watch register write | → | instruction taking exception | `Watch` | **eret** |
| **tlbw** | → | instruction fetches | TLB | **eret** |
| `Compare` | → | instruction not seeing timer interrupt | Timer interrupt | +4 cycles |
| **cache** | → | instruction fetches | Instruction cache | **eret** |

*Table 11.1: CP0 Hazards and required action*

# 12. References


MIPS64 The ‘‘MIPS64 Architecture Reference Manual’’. This comes in three volumes. A fourth volume is required to define release 2 of the architecture:

| *Vol* | *Subtitle* | *Document No* | *Rev* |
|---|---|---|---|
| volume 1 | Introduction to the MIPS64 Architecture | MD00081 | 1.00 |
| volume 2 | The MIPS64 Instruction Set | MD00085 | 1.00 |
| volume 3 | The MIPS64 Privileged Resource Architecture | MD00089 | 1.00 |
| | Release 2 Architecture Changes | MD00145 | 1.50 |

The references [MIPS64V2] and [MIPS64V3] are to volume 2 and volume 3.

EJTAG MIPS Technologies’ ‘‘EJTAG Specification’’, document number MD00047, v2.61.

SEEMIPSRUN ‘‘See MIPS Run’’, Dominic Sweetman, published by Morgan Kaufmann, ISBN 1−55860−410−3.

# Appendix A: CP0 registers by name and number

| *Register No./Set* | *Register Name* | *Function* | *See ref/ section* |
|---|---|---|---|
| 0.0 | **Index** | Index into the TLB array | [MIPS64] |
| 1.0 | **Random** | Randomly generated index into the TLB array | |
| 2.0 | **EntryLo0** | Low-order portion of the TLB entry for even-numbered virtual pages | |
| 3.0 | **EntryLo1** | Low-order portion of the TLB entry for odd-numbered virtual pages | |
| 4.0 | **Context** | Pointer to page table entry in memory | |
| 5.0 | **PageMask** | Control for variable page size in TLB entries | §8, p.16 |
| 6.0 | **Wired** | Controls the number of fixed ("wired") TLB entries | [MIPS64] |
| 8.0 | **BadVAddr** | Reports the address for the most recent address-related exception | |
| 9.0 | **Count** | Processor cycle count | §6.1, p.10 |
| 10.0 | **EntryHi** | High-order portion of the TLB entry | [MIPS64] |
| 11.0 | **Compare** | Timer interrupt control | |
| 12.0 | **Status** | Processor status and control | §6.2, p.10 |
| 13.0 | **Cause** | Cause of last general exception | [MIPS64] |
| 14.0 | **EPC** | Program counter at last exception | |
| 15.0 | **PRId** | Processor identification and revision | Figure 3.1, p.7 |
| 16.0<br>16.1 | **Config**<br>**Config1** | Configuration register<br>Configuration register 1 | Table 3.1, p.6<br>[MIPS64] |
| 17.0 | **LLAddr** | Load linked address | |
| 18.0 | **WatchLo** | Watchpoint address | Figure 10.1, p.18 |
| 19.0 | **WatchHi** | Watchpoint control | |
| 20.0 | **XContext** | Extended Addressing Page Table Context | [MIPS64] |
| 23.0 | **Debug** | EJTAG Debug register | [EJTAG] |
| 24.0 | **DEPC** | Program counter at last EJTAG debug exception | |
| 25.0<br>25.1 | **PerfCtl**<br>**PerfCnt** | Performance counter control<br>Performance counter | Figure 10.2, p.19<br>§10.2, p.19 |
| 26.0<br>26.1 | **DErrCtl**<br>**IErrCtl** | Parity/ECC error control and status | §5.2, p.9 |
| 27.0 | **CacheErr** | Cache parity error control and status | |
| 28.0<br>28.2<br>28.1<br>28.3 | **ITagLo**<br>**DTagLo**<br>**IDataLo**<br>**DDataLo** | Low-order portion of cache tag interface<br><br>Low-order portion of cache data interface | Figure 7.1, p.11<br>Figure 7.2, p.14<br>DIAG |
| 29.0<br>29.1<br>29.3 | **ITagHi**<br>**IDataHi**<br>**DDataHi** | High-order portion of cache tag interface<br>High-order portion of cache data interface | Figure 7.1, p.11<br>DIAG |
| 30.0 | **ErrorEPC** | Program counter at last error | [MIPS64] |
| 31.0 | **DESAVE** | EJTAG debug exception save register | [EJTAG] |

## By name

| *Register Name* | *Number* | *Register Name* | *Number* | *Register Name* | *Number* | *Register Name* | *Number* |
|---|---|---|---|---|---|---|---|
| **BadVAddr** | 8.0 | **DEPC** | 24.0 | **IDataHi** | 29.1 | **PerfCtl** | 25.0 |
| **CacheErr** | 27.0 | **DESAVE** | 31.0 | **IDataLo** | 28.1 | **Random** | 1.0 |
| **Cause** | 13.0 | **DErrCtl** | 26.0 | **IErrCtl** | 26.1 | **Status** | 12.0 |
| **Compare** | 11.0 | **DTagLo** | 28.2 | **ITagHi** | 29.0 | **WatchHi** | 19.0 |
| **Config1** | 16.1 | **Debug** | 23.0 | **ITagLo** | 28.0 | **WatchLo** | 18.0 |
| **Config** | 16.0 | **EPC** | 14.0 | **Index** | 0.0 | **Wired** | 6.0 |
| **Context** | 4.0 | **EntryHi** | 10.0 | **LLAddr** | 17.0 | **XContext** | 20.0 |
| **Count** | 9.0 | **EntryLo0** | 2.0 | **PRId** | 15.0 | | |
| **DDataHi** | 29.3 | **EntryLo1** | 3.0 | **PageMask** | 5.0 | | |
| **DDataLo** | 28.3 | **ErrorEPC** | 30.0 | **PerfCnt** | 25.1 | | |

## By Function

| | | |
|---|---|---|
| *Basic status* | **Status** | 12.0 |
| *Exception control* | **BadVAddr** | 8.0 |
| | **EPC** | 14.0 |
| | **Cause** | 13.0 |
| *Parity/ECC control* | **CacheErr** | 27.0 |
| | **DErrCtl** | 26.0 |
| | **IErrCtl** | 26.1 |
| | **ErrorEPC** | 30.0 |
| *Timer* | **Compare** | 11.0 |
| | **Count** | 9.0 |
| *Configuration* | **PRId** | 15.0 |
| | **Config** | 16.0 |
| | **Config1** | 16.1 |
| *TLB maintenance* | **Context** | 4.0 |
| | **XContext** | 20.0 |
| | **EntryHi** | 10.0 |
| | **EntryLo0** | 2.0 |
| | **EntryLo1** | 3.0 |
| | **PageMask** | 5.0 |
| | **Index** | 0.0 |
| | **Random** | 1.0 |
| | **Wired** | 6.0 |
| *Cache* | **ITagHi** | 29.0 |
| | **ITagLo** | 28.0 |
| | **DTagLo** | 28.2 |
| | **IDataHi** | 29.1 |
| | **IDataLo** | 28.1 |
| | **DDataHi** | 29.3 |
| | **DDataLo** | 28.3 |
| *EJTAG* | **Debug** | 23.0 |
| | **DEPC** | 24.0 |
| | **DESAVE** | 31.0 |
| *debug* | **PerfCtl** | 25.0 |
| | **PerfCnt** | 25.1 |
| | **WatchHi** | 19.0 |
| | **WatchLo** | 18.0 |
| *Load-linked* | **LLAddr** | 17.0 |