Programming the MIPS32® 1004K™ Coherent Processing System Family

Document Number: MD00638
Revision 01.20
January 21, 2011

Chapter 1

Introduction

The MIPS32® 1004K™ Coherent Processing System is a high performance coherent multiprocessing system consisting of 1-4 32-bit MIPS32 1004K™ processor cores and an optional coherent I/O port. The 1004K core is a 32-bit MIPS32 CPU. In typical 65nm process technology it runs at up to 800MHz with a 8-9-stage pipeline. But what is special about the 1004K core are the following features:

- *Cache Coherence:* The core is designed to be used inside of a Coherent Processing System that may include multiple cores as well as an adapter for coherent I/O devices (think DMA). The data cache controller communicates with a Coherence Manager and ensures that data generated on one core (or input device) is made visible to the other cores (and output devices) - that is, the I/O and various caches are kept coherent with one another. Programming considerations for ensuring correctness and maximum performance on multiple cores could fill many volumes and is unfortunately not covered in this document.

- *The MIPS® MT ASE*: The multithreading ASE (“application-specific extension” to the MIPS architecture). It’s a modest addition to the instruction set, but a profound change to the CPU, which can now run multiple threads concurrently. The set of software-visible resources devoted to one thread are known as a *TC*. The MT ASE allows for two multithreading models which are very different for software:

  – Multiple *Virtual Processing Element*s (*VPE*s) in a CPU: each “VPE' has at least one TC together with its own copies of everything required to make it just like an independent MIPS CPU. Your 2-VPE (or more) core seems to software just like a 2-CPU “SMP” multiprocessor: indeed, it can run SMP software - software which knows nothing about MIPS MT - without requiring any CPU-related changes.

  – Multiple concurrent threads running within one VPE, usable by software which knows about MIPS MT. These multiple threads are relatively cheap, because they’re equipped only with the resources necessary to run user-level programs (but they share a lot of OS-controlled resources.)

  The 1004K core only supports the first model. This model is quite seamless since SMP software would typically be used for distributing work across the multiple cores within the Coherent Processing System.

  Much of this manual won’t make any sense until you get your head round multithreading, so unless you’re thoroughly familiar with it already you should acquaint yourself with Chapter 2, “The MIPS® MT ASE - Multithreading the RISC way” on page 17.

- *DSP ASE*: this adds a lot of new computational instructions with a fixed-point math unit crafted to speed up popular signal-processing algorithms, which form a large part of the computational load for voice and imaging applications. Some of these functions are ‘‘SIMD” - they might, for example, do two math operations at once on two 16-bit values packed into one 32-bit register.

  There’s a guide to the DSP ASE in Chapter 9, “The MIPS32® DSP ASE” on page 127 and the formal specification is [MIPSDSP].

1.1 Readership

This document is for programmers who are already familiar with the MIPS® architecture and who can read MIPS assembler language (if that's not you yet, you'd probably benefit from reading a generic MIPS book, see Appendix A, "References" on page 179).

More precisely, you should definitely be reading this manual if you have an OS, compiler or low-level application which already runs on some earlier MIPS CPU, and you want to adapt it to the 1004K core. So this document concentrates on where a MIPS 1004K family core behaves differently from its predecessors. That's either:

- Behavior which is not completely specified by Release 2 of the MIPS32® architecture: these either concern privileged operation, or are timing-related.
- Behavior which was standardized only in the recent Release 2 of the MIPS32 specification (and not in previous versions). All Release 2 features are formally documented in [MIPS32][1], and [MIPS32V1] contains a brief summary.

  But the summary is too brief to program from, and the details are widely spread; so you'll find a shortform presentation of the changes here in Section D.2 "User-level changes with Release 2 of the MIPS32® Architecture".
- Details of timing, relevant to engineers optimizing code (and that very small audience of compiler writers).

This manual is intentionally much more focussed and therefore smaller than the full [SUM] manual. It does leave some material out; if you need to write processor subsystem diagnostics, this will not be enough! If you want a very careful corner-cases-included delineation of exactly what an instruction does, you'll need [MIPS32V2]... and so on.

For readability, some MIPS32 material is repeated here, particularly where a reference would involve a large excursion for the reader for a small saving for the author. Appendices mention every user-level-programming difference any active MIPS software engineer is likely to notice when programming the 1004K core.

1.2 In this chapter

In the remainder of this chapter you'll find:

- Section 1.3, "Chapter summary": what's in the chapters (hot links if you're reading online).
- Section 1.4, "Typographical conventions": a manual like this is made easier to read (though perhaps not made more beautiful) if we use typographical conventions so you can recognize machine registers, instructions and so on. Here's what they look like.
- Section 1.6 "Finding information in this manual": how to find things in here, whether you're reading online or on paper.
- Section 1.7, "Specification summary": a terse summary of facts and figures.

[1] References (in square brackets) are listed in Appendix A, "References" on page 179.

1.3 Chapter summary

- Chapter 2, “The MIPS® MT ASE - Multithreading the RISC way” on page 17: about the MIPS Multi-Threading instruction set extension (“ASE”).
- Chapter 3, “How the 1004K™ core implements multi-threading” on page 41: implementation options and more details.
- Chapter 4, “Initialization and identity” on page 55: setting up the 1004K core, including its multi-threading system.
- Chapter 5, “Memory map, caching, reads, writes and translation” on page 69: all about memory accesses and translation.
- Chapter 7, “Kernel-mode (OS) programming and Release 2 of the MIPS32® Architecture” on page 105: use of “hazard barriers”, the advanced interrupt system, shadow registers and power management.
- Chapter 10, “1004K™ CPU features for debug and profiling” on page 142: EJTAG debug unit, watchpoints and performance counters.
- Chapter 6, “Programming the 1004K™ CPU in user mode” on page 97: on tuning code specifically for the 1004K core family.
- Chapter 8, “Floating point unit” on page 117: a software view of the (optional) 64-/32-bit floating point unit.
- Chapter 9, “The MIPS32® DSP ASE” on page 127: the instruction set extension for faster media algorithms.

Appendices:

- Appendix A, “References” on page 179: further reading.
- Appendix B, “Glossary” on page 181: a glossary of terms which may be unfamiliar (particularly relating to multi-threading).
- Appendix C, “CP0 register summary and reference” on page 185: functionally orientated index to the 1004K core’s “co-processor zero” registers and fields, and contains descriptions of all the registers not already included in one of the other chapters.
- Appendix D, “MIPS® Architecture quick-reference sheet(s)” on page 197: handy guide to easily-forgotten data on MIPS.
- Appendix E, “” on page 201: for this document.

1.4 Typographical conventions

CPU register names are in *oblique monospace*. *Co-processor zero* (*CP0*) registers fields are shown after the register name in brackets, so the interrupt enable bit in the *Status* register appears as *Status[IE]*. 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.

The acronym *CP0* in the paragraph above is a word defined in Chapter B, “Glossary” on page 181 and shows up in *italics* - but if you’re reading on-line it also shows up as *blue*, showing that it’s a link which you can click to get to the definition.

References to other manuals are collected together in Appendix A, “References” on page 179 and look like this [MIPS32].

Instruction mnemonics and assembler code fragments are set in **`bold monospace`**, core interface signal names in *small italics*, and C or other programming language constructs in `monospace`.

To use register and field names in your program, you’ll need a C header file or something similar. It’s probably better and easier not to write your own: see [m32c0.h] and [mt.h].

1.5 Register diagrams and field descriptions

It’s a tradition of MIPS CPUs that most control and status information is passed through registers - the most numerous are the “CP0” registers used for kernel-level CPU control operations, but there are also memory-mapped registers in the debug unit and to control special memory arrays. All of them are 32 bits wide.

Many of the registers are broken up into multiple fields with substantially independent meanings and effects. Any register which is not simply a 32-bit number comes with a register “figure”, and there’s a list of figures at the start of this manual. The register figures are growing extra information in this version of the manual:

- We’re introducing color-codes to identify fields. Fields which you can write, have some hardware effect and read back the same are regarded as “standard” and have a white background. But the background color tells you which fields are read-only (green), which are zero or “X” (gray), are purely for software use (blue-green), which are *not* just write-and-read-back (yellow), or are reserved and where use might be dangerous (red):

read-only (green)	zero/X (gray)	software-only (blue/ green)	not just write-back (yellow)	reserved, take care (red)

  If you’ve printed this manual in black-and-white, those will all look much the same, sorry!

- Register diagrams may carry a third row (below the field descriptions in the boxes) which tell you about any value guaranteed to be in the register after a hardware reset. Those values will always be described separately in the field descriptions, and careful programmers will probably avoid relying on them wherever they can.

1.6 Finding information in this manual

If you’re reading this manual on-screen, text shown in blue is a hot-link; click on the text to go to the section, figure or table referenced. The chapter index and lists of tables and figures at the start of the book is click-through too.

All the special *Co-processor zero* (*CP0*) registers are listed in Appendix C, “CP0 register summary and reference” on page 185. That appendix has the registers listed by name, by number and by function. The by-number table has hot-links to other sections where each is mentioned - and for those reading on paper, all those links have page numbers.

1.7 Specification summary

The 1004K core is provided as a synthesizable package, and customers have considerable freedom to customize it. But all 34K cores share these:

- *CPU architecture*: compliant to Release 2 of the MIPS32 Architecture [MIPS32].
- *Multi-threading*: as defined by the Multithreading extension to the MIPS32 architecture as specified by [MIPSMT].

  The 1004K core can be synthesized to be able to run nine concurrent threads (9 *TC*s) in up to two "virtual processors" (2 *VPE*s).

  It may be equipped with a bank of *Inter-Thread Communication storage* (*ITC*) locations, following the recommendations of [MIPSMT].
- *DSP-orientated instruction set*: it implements the DSP extension to the MIPS32 architecture, see [MIPSDSP].
- *MIPS16e*™: the 16-bit instruction set option for compact code, see [MIPS16e].
- *8-9-stage pipeline*[2]: a sophisticated branch prediction unit keeps the CPU efficient, even when it's only running one thread.
- *Separate I- and D-caches*: 4-way set associative. The SoC designer may choose from 8, 16, 32 or 64Kbytes size for each cache (and can even omit the I cache). Parity checking in the cache is optional.

  Caches are non-blocking, and both allow for hit-under-miss and miss-under-miss - the I-cache uses that to allow a cache-hitting thread to continue even though an I-cache refill is pending for some other thread.

  The D-cache is write-back (memory regions may also be configured as a special "uncached accelerated" write mode). You can lock data into the caches.
- *OCP system interface*: industry-standard interconnect.

SoC Builder's Optional features

Some features are provided only at the option of the SoC integrator, and may depend on separate licensed material from MIPS Technologies:

- *L2 (secondary) cache:* you can configure your 1004K core with MIPS Technologies' L2 cache between 128Kbyte and 1Mbyte in size. Full details are in [L2CACHE], but programming information is in Section 5.4 "Caches" of this manual.
- *CorExtend™ user-defined instructions*: the 1004K Pro Series™ core family allows you to add custom instructions as described in [CorExtend].
- *Floating point unit*: fitted to 1004Kf™ cores, with 32 full 64-bit floating point registers.
- *Fixed mapping MMU*: reduces core size when a TLB is not required.
- *Instruction- or data-side "scratchpad" memory*: each can be up to 1Mbyte of high-performance on-chip memory, which can be dual-ported to the OCP interface for "push" I/O architectures.
- *EJTAG debug unit*: on-chip debug resources, summarized in Section 10.1, "EJTAG on-chip debug unit".

[2] Single TC configurations enable a bypass of a thread selection stage to get down to 8 stages, otherwise it'll be 9. When executing MIPS16e instructions, it'll be 11 stages

- *Power-management options*: summed up in Section 7.5, "Saving Power" below.
- *OCP L2 extensions*: to allow front-side L2 cache.

Refer to [INTGUIDE] for full details about the options.

1.8 Pipeline and implementation

In programming documents about MIPS Technologies cores you'd usually find a section which describes the pipeline, at least at a broad level useful for programmers. With the 1004K core that is hard to describe without knowing something about multi-threading so we've moved it to Section 3.1, "The 1004K™ core pipeline and multithreading" below.

Chapter 2

The MIPS® MT ASE - Multithreading the RISC way

We use “MT” for “multi-threading”. So what does a MIPS architecture CPU do to run multiple threads concurrently? That question is one about “architecture” - the corresponding “how does the 1004K core run multiple threads?” question is about implementation, and is answered below in Chapter 3, “How the 1004K™ core implements multithreading” on page 41.

In this chapter:

- Section 2.1, "What’s a thread and its context?": basic definitions.
- Section 2.2, "Why multi-threading?": motivation.
- Section 2.3, "Different kinds of multi-threading: TCs and VPEs": we offer two levels of multi-threading in one CPU.
- Section 2.4, "When can’t threads run?": and what they’re doing when stopped.
- Section 2.5, "Thread-scheduling decisions and the policy manager": what happens and what influence can you have.
- Section 2.6, "Multithreading, exceptions and interrupts": interrupts and other exceptions in the MIPS MT CPU.
- Section 2.7, "Multithreading, non-blocking loads and stores, and gating storage"
- Section 2.8, "MIPS® Multithreading ASE - new instructions"
- Section 2.9, "Multithreading ASE - CP0 (privileged) registers": understanding multi-threading in fine detail.

Why multi-threading takes a lot of thinking about

Any form of concurrency makes your head hurt. Our brains are doubtless extremely parallel: we can talk on a cellphone and drive with only a 50% increase in our chance of crashing. But our ability to reason correctly is distinctly sequential, and so far we have not bred a race of super-kids who can write explicitly parallel software.

Multi-tasking software has been successfully understood by dividing it into sequential chunks ("threads", though a more precise definition follows) which communicate and synchronize with each other only in carefully controlled ways. You can then unleash a flock of threads and allow them to evolve separately. Programmers find it almost impossible to keep track of what every thread is doing at any one time - but with simple-enough rules about the interactions, the system will still work.

The multithreading CPU pushes thread concurrency down to the hardware level, so you should expect to find it somewhat mystifying from time to time. To really understand multi-threading and the 1004K core you need to be able to switch between a software-orientated threads-eye-view (where threads are internally sequenced and other threads are happening somewhere else) and a hardware engineers CPUs-eye-view (where everything happens in sequence along the pipeline). This is difficult, but we hope not impossible. This chapter takes the “thread” viewpoint, and the next chapter stays closer to the hardware.

2.1 What’s a thread and its context?

There are a couple of critical phrases and acronyms which it’s useful to define carefully before we start:

- *Thread*: a set of computer instructions read and activated in their programmed order.

  Operating systems most often use the word “thread” specifically for application-software visible threads scheduled by the OS. But our wider definition means that any piece of software must have at least one thread.

  By this definition something like an interrupt handler (which is not reached as a result of normal program flow) counts as a thread in its own right. This more general definition of "thread" seems to be a more logical starting point for describing multi-threading hardware.

- *Thread context*: you might want to consider the complete state of a running thread, enough so you could restart it successfully. But for our purposes we’re particularly interested in the part of the state which gets stored inside the CPU - what [MIPSMT] calls the “thread context”. The thread context always (of course) includes the *Program Counter (PC)* and the general-purpose registers. There are some good justifications for narrowing our focus down to the state held in the CPU:

  1. We don’t need to encompass the thread’s data stored in memory, because we know how to share memory already (for OS-defined threads, for example);

  2. We don’t include state which is inherently inaccessible to this particular instruction stream - so kernel-only readable CP0 registers are invisible to a user-privilege thread;

  3. We don’t include state which is logically unnecessary, and just kept for efficiency - for example, cache contents, which generally make no difference to the underlying memory image.

  With this definition, what is included in the thread context varies according to what sort of software is running. For a Linux interrupt handler on a conventional MIPS architecture CPU the CP0 registers are part of the thread state, but for a Linux application thread they’re not visible.

You could have found the definitions of *Thread* and *Thread context* in Appendix B, “Glossary” on page 181 below. Any word or phrase in blue (or slightly faint in real black-and-white print) is probably explained. If you’re reading online and it’s blue, it will link to its definition: try it.

2.2 Why multi-threading?

Traditionally, a CPU only held one thread’s context (one PC, one set of registers). Operating systems providing multiple threads held all the state for the non-running threads in OS-specific data structures.

But MIPS MT CPUs are equipped with more than one PC and register set so they can hold more than one thread’s context.

There’s more than one reason why you might want to build a multithreading CPU. For MIPS MT the main motivation is to build a CPU which can continue to do useful work when some computation is held up for a period of a few to some hundreds of CPU cycles - typical of cache misses and some other interactions in embedded systems. Such a hold-up is too short to allow an OS to borrow the CPU to do something else (the OS thread-switch overhead is itself probably 100 cycles or more). But in many workloads such hold-ups are frequent enough that the CPU spends half its time waiting for data.

A multithreading CPU can keep other threads making progress when one thread is held up. If (as is commonly the case these days) the real workload is already split into multiple threads, that can turn into extra application performance without modifying application code.

The extra thread state storage (mostly the register file) only represents a fraction of the gate count of a CPU, so this extra performance has cost only a small increment in area and complexity. That's why in 2005 everyone wants to do multithreading.

2.3 Different kinds of multi-threading: TCs and VPEs

In some ways the simplest thing to do is to replicate every software-visible piece of CPU state. Then your multithreading CPU will look pretty much like two CPUs which happen to share memory, creating a "virtual multiprocessor" (*VSMP*). That's what Intel's newer multithreading x86 processors do; you can drop a Linux kernel designed for a two-way multiprocessor onto such a CPU and it just works. It's an easy way to get a software market for a new technology.

But performance-critical embedded applications are those where the multithreading is an explicit part of the system design - we'll call it "explicit multithreading" or *EMT*. EMT is new, so we don't need to offer backward compatibility. An EMT application does not need the whole CPU replicated; it can manage with what is visible to user-level programs - the PC, GPRs and a little more.

The original and ingenious trick in the MIPS MT architecture is that you have a choice of either model, and can even do both in the same CPU at the same time. So a MIPS MT CPU has multiple *TC*s (the acronym started out as *Thread context*), but also provides for more than one *VPE* ("VPE" started out as a *Virtual Processing Element*.) A TC provides the minimum required to do explicit multithreading, while one or more TCs with their own VPE really look like an independent CPU, enough to provide a congenial home for software which doesn't really want to know about MIPS MT - perhaps even a non-MT-aware legacy operating system.

2.3.1 How an MT CPU's hardware uses TCs and VPEs

Each instruction being run by an MT CPU has a TC number. Whenever the instruction accesses some state - reads or writes a general-purpose register, for example - it uses its TC number to extend the register-number field which is already defined inside the instruction. An instruction sees a different set of registers depending on the TC number: it's very simple, and it just works.

It's not quite that simple on a MIPS architecture CPU, because of the TC/VPE trick mentioned above. So this instruction might be for TC #5 (it uses general purpose registers from the fifth bank) but VPE #1 (it gets most of its CP0 registers from the first bank). Again, this should just work. What's more complicated, of course, is to get those CPU resources working which can't simply be reduced to registers. But that's not architecture, it's implementation, and described in Chapter 3, "How the 1004K™ core implements multi-threading" on page 41 below.

2.3.2 CPU resources and registers shared between all threads

Many of the CPU's resources are not replicated for MIPS MT, just used by whichever TC is identified by the instruction accessing the resource. They include:

- *Caches*: the cache's contents are just like memory (only faster) and unproblematic. On a CISC CPU the cache is usually completely invisible to running software, and there's no issue at all about multiple threads - but MIPS architecture CPUs generally need the OS to intervene in the caches at some points.

  The MIPS MT ASE requires that the writeback and invalidate **`cache`** instructions used by real OS' when running are multi-threading safe. Cache manipulations may be independently mixed by two VPEs[3] without immediate harm; even if one VPE invalidates a cache entry from right under the feet of another one, everything should keep working - the consuming VPE will either get the old copy (which it was happy with) or cache-miss and pull in a new one (which should be just the same data).

  However, arbitrary re-initialization of a cache already in use by another VPE will not be safe; writeback data could be lost. Programs running on separate VPEs would probably be well-advised to get cache initialization done by a thread running alone before other VPEs are enabled.

  With a multithreading workload, cache performance could suffer; multiple threads will probably produce a larger and more diverse "working set" of active memory regions. However, a cache works well (or not) when optimizing repeated accesses over spans of code executing hundreds of thousands to millions of instructions. During that time which even a single-threaded workload will climb all over application and OS space. The 1004K core's caches are already 4-way set associative, which should be enough to minimize misses caused by overlapping hotspots of several concurrent threads. Our measurements to date back that up.

- *Main pipeline*: each of the 1004K core's main pipeline stages just serve the TC associated with the current instruction. No problem.

- *The TLB (sometimes)*: the MIPS MT ASE allows the TLB entries to be shared between all VPEs, or partitioned between VPEs. The 1004K core can be configured to do either (to share the TLB, set *MVPControl[STLB]* to 1.)

  If the TLB is not shared, it is partitioned by hardware so each VPE sees its own independent array of entries.

  When the TLB is shared, there's a problem of managing concurrent access by the two VPEs. It's up to OS software to control concurrent access by OS maintenance routines. But that still leaves the risk that one VPE's maintenance software will collide with another VPE's TLB refill exception handler: see Section 4.3.4, "Sharing and not sharing the TLB" for how that's avoided.

- *Basic configuration registers*: in a highly adaptable design like the 1004K core the initialization software needs to know the full resource complement of the CPU, or it can't know how to share it between the VPEs.

  The registers *MVPControl* and *MVPConf0-1* allow software to see what resources are provided CPU-wide, and these registers are not replicated per-VPE.

This is more implementation than architecture, but some software-invisible resources are also shared. Notably, the 1004K core's "branch history table" (BHT) in the instruction fetch unit is shared. That seems quite wrong: the branch histories of different threads are certainly likely to be different. But the BHT was only statistically correct anyway; the branch history is only recorded in entries indexed by some modest number of low virtual address bits. Even in a conventional single-thread CPU, different branches could map onto the same entry and cause confusion (and thus lower the prediction accuracy) - but there are enough different entries that this relatively rarely happens. Having multiple threads doesn't really make it much worse, and the BHT should continue to perform well in typical applications.

[3] The CP0 registers used with the **`cache`** instruction are only replicated per-VPE, so EMT code must take care to avoid re-entry into cache management functions by other threads.

2.3.3 CPU resources and registers replicated per-TC

Some state needs to be independently kept for each TC, including:

- *Program counter and general purpose (integer) registers*: the TC’s program counter can be seen and adjusted (when the TC is halted, otherwise it’s a moving target) in *TCRestart*. The architecture does not define what you’ll get if you read your own *TCRestart*; probably some "historical value".

  Each TC, of course, has its own set of 32 general purpose registers. It also needs its own copies of the accumulator registers in the multiply-divide unit (*hi*/*lo*), and the extra accumulator registers and control register provided as part of the DSP ASE described in Chapter 9, “The MIPS32® DSP ASE” on page 127.

- *Privilege state*: some TCs (sharing a VPE) may be in the kernel while others are running user-mode software. So each TC has its own copy of the user-mode/kernel-mode flags *Status[KSU]*. *TCStatus[TKSU]* provides a convenient per-TC view of the same flags. Each TC gets a copy of the *TCContext* register too: it has no hardware significance, but provides a useful scratch register for the OS to keep some key thread identifier.

- *Address space*: we don’t want to insist that all TCs which share a VPE must execute in the same address space. Different address spaces in MIPS architecture CPUs are managed by only returning TLB translations for virtual addresses when they’re presented together with the right "ASID" value, an arbitrary 8-bit token held in *EntryHi[ASID]* while the system runs.

  So each TC also has its own copy of the *EntryHi[ASID]* field - the same field is accessible as *TCStatus[TASID]*.

- *Performance counters*: This isn’t specified by the architecture. In fact, earlier versions of the core had four counters that were shared by all TCs and VPEs. Later, this was changed to provide two counters per TC to provide more counters in larger configurations and to eliminate the need to arbitrate between VPEs for access. Each counter can count events from any TC as well as events that are per-VPE (few) or pre-processor (many)

- *Access to co-processors*: the 1004K core’s FPU - when fitted - is built with just one set of registers. That makes sense because the registers in the floating point unit already occupy a lot of logic space, and the 1-register-set FPU design is identical to that used in the 24K™ core family. But it means that the FPU can’t be used by multiple concurrent threads.

  Some other co-processors might have one set of data registers per TC, supporting arbitrary multi-threading.

  In the MIPS architecture you can’t use any co-processor unless you first turn on the corresponding *Status[CUx]* bit in the status register.   MIPS MT uses that to provide a mechanism to share the co-processors, detailed in the notes to Figure 2.2 below. As part of that mechanism the *Status[CU3-1]* bits are also visible at *TCStatus[TCU3-1]*.

- *Which VPE we’re using*: a TC must know which VPE is belongs to, or it can’t get at the right copy of the per-VPE registers. The VPE affiliation is readable and writable in *TCBind[CurVPE]*. (Each VPE also has a distinct number readable at *EBase[CPUNum]*, to allow seamless use of multi-CPU software on multiple VPEs.)

- *TC halted*: think of this as "TC anesthetized" - it stops the TC from wriggling around when under surgery, or even just close inspection. It occupies its own 1-bit register *TCHalt* so it can be set and cleared atomically.

  While this is set the TC is frozen: won’t run, can’t be picked by **`fork`**. The architecture abhors the idea of a halted thread being half-way through a synchronization access, and any pending load/store to *Gating Storage* will be rolled back when this bit is set. From a hardware point of view the gating storage access is aborted; but unless you do something special to stop it the access will be quietly retried once the OS is finished with its maintenance and clears *TCHalt*.

- *TC interrupt-exempt*: set *TCStatus[IXMT]* to mean this TC will never be picked to handle an interrupt exception (even if that means the interrupt is completely ignored).

- *Per-TC flags*: there are also bits to control the ability of **`fork`** to seize a “free” TC and make it run a new thread, and for other purposes. See the description of **`fork`** in Section 2.8, "MIPS® Multithreading ASE - new instructions" and the notes on Figure 2.2.

- *Debug state*: the single-step bit *Debug[SSt]* is replicated per-TC, for fine debugger control. The debugger is also given a control bit *Debug[OffLine]* which it can use to prevent TCs other than the one under debug from springing into life during single-step or when running a thread to the next breakpoint.

2.3.4 CPU resources and registers replicated per-VPE

We want a TC running alone in a VPE to be a MIPS32-compliant processor in its own right, so each VPE replicates all the CP0 registers required by release 2 of the MIPS32 specification (a few read-only registers are in fact shared between VPEs on the same CPU, but they’re read-only, so who’s to know?)

So what is replicated?

- *State related to exceptions*: MIPS architecture experts will recall that you enter exception mode by taking an exception, and remain in it until you either return with an **`eret`** or (more common in a complicated OS) you carefully clean up exception-dependent information and then manually clear *Status[EXL]*.

  The MIPS MT architects determined that only one TC from a VPE is allowed to be in exception mode at any one time - when one TC takes the exception, its VPE siblings are suspended until the first TC clears *Status[EXL]*. To do otherwise would require a lot of extra replicated state, and would lead to some nasty concurrency hazards.

- *Interrupt system and interrupts*: interrupt signals to the chip are wired to VPEs separately (a reasonable strategy may be to wire all the VPEs in parallel to the same inputs, but that’s an SoC designer’s decision).

  The interrupt management fields in the *Cause* and *Status* registers are all per-VPE.

- *Cache management registers*: all the cache operation staging registers are per-VPE. In fact, most of the CP0 registers are per-VPE.

- *The TLB (sometimes)*: on the 1004K core the TLB may either be shared, or partitioned invisibly so that two VPEs each think they have their own dedicated chunk of the TLB[4].

- *The EJTAG debug unit*: the physical unit may or may not be replicated, but the registers in its CP0 software interface (*DEPC*, *DESAVE* and *Debug*) are replicated per-VPE.

  In debug mode all TCs other than the one running the debugger are suspended, regardless of VPE affiliation. Moreover, the TC in debug mode continues to run even if it is otherwise marked as halted, not-allocated etc. More details in Section 10.1.2, "Debug mode".

2.4 When can’t threads run?

A CPU can be compliant to the MIPS MT ASE without being committed to any particular thread-scheduling algorithm - the decision as to which thread’s instruction to pick next is implementation-dependent. But that level of

[4] The amount of the TLB awarded to each VPE is configurable when your core is synthesized. Ask your hardware engineer.

abstraction is difficult, so let’s make some working assumptions - which will, happily, turn out to be correct for the 1004K core.

Some implementations permit customizable hardware outside of the core to influence the CPU to favor one TC over another when deciding what instruction to run next; see Section 2.5, "Thread-scheduling decisions and the policy manager" below.

But before worrying about that, let’s look at something simpler. A practical CPU might run instructions in turn ("round-robin") from each live thread. But what about that weasel word "live"? When can a thread *not* make progress? Well, it can be:

- *Waiting for memory data*: most often, to resolve a cache miss (for of the order of 50 cycles) - making use of this idle time is the first motivation for contemporary multithreading.

  Or this might also be an uncached read of some device-register data (of the order of 100-500 cycles) - particularly relevant to embedded applications.

- *Blocked on read/write “gating storage”*: we envisage that multithreading applications are likely to use special memory locations where the wait-for-transfer is used as a deliberate way of matching the speed of the software to the arrival of data either from other threads, or some direct hardware source/sink. Waits of this kind may extend for thousands of cycles. So the MIPS MT *ASE* describes how some memory locations are accessed according to special rules which make them *Gating Storage*, and describes a particular application of gating storage to optional *ITC* locations. See Section 3.3, "Inter-thread communication storage (ITC)" for the facility provided by the 1004K core.

- *Blocked on an "interrupt-like" external signal*: a thread which waits for a particular hardware signal is an obvious multithreading analogue of an interrupt handler, and likely to be useful. You’ll see how the MIPS MT **yield** instruction can be used for that purpose.

- *Halted - closed for maintenance*: there are bound to be things the OS wants to do with TCs which can’t be done while it’s live, and each TC comes with a "Halt" button in the *TCHalt* register.

- *Not "allocated":* the MT system includes the **fork** instruction, which provides a very lightweight way of starting a new thread - potentially, it’s even usable from user-mode in a protected OS. An OS obviously can’t simply relinquish control of thread scheduling, but it can arrange to provide a pool of ‘‘free’’ threads which **fork** can use - they’re a bit like taxis waiting at a taxi-rank for customers. The TCs “at the rank” are prevented from running code by having their *TCStatus[A]* (“allocated”) bit clear. If a system doesn’t use **fork**, then it must take care to set the allocated bit explicitly on any TC which is to run.

- *Affiliated to an unactivated VPE*: that is, one with *VPEConf0[VPA]* zero.

- *Asleep after executing a wait instruction*: in which case it won’t awake until its VPE gets an interrupt (it doesn’t matter which TC runs the interrupt code, all TCs are woken from their sleep).

- *Suspended - temporarily inhibited to avoid some concurrency problem*: for example we’ll see that a VPE becomes "single-threaded" while it is handling exceptions, so that implicitly suspends all the VPE’s other TCs. OS software can achieve a similar effect using instructions such as **dmt** (stop all other threads with the same VPE affiliation) and **dvpe** (stop all other threads, even in different VPEs).

- *“Offlined” by a debugger*: using *Debug[OffLine]*, typically so the debugger can isolate another thread for test.

In this manual we’ll try to consistently use the word *stopped* for a thread subject to any of the conditions above - and by analogy, we’ll use the same adjective to describe the TC which is executing the thread. The opposite of "stopped" is *live*.

We’ll distinguish a stopped thread as either:

- *Stalled*: waiting for a condition which could be experienced by a program on a single-threaded CPU - that includes waiting for data from a cache miss or an uncached read, OR:
- *Blocked*: waiting for something other than the above. That’s some deliberate multithreading synchronization by **`yield`**, a gating storage read, or explicitly stopped as a result of software activity.

  The blocked state is new with MIPS MT. The nearest thing that a thread on a non-MT MIPS CPU can come to "blocked" is when the CPU is asleep after executing **`wait`**.

For blocked threads we’ll use *halted*, *suspended* and *asleep* in the specific senses above. The use of these terms is compatible with the formal specification [MIPSMT], though that uses *running* to instead of live. In the formal specification “running” means either live or waiting for a normal read/write.

Regardless of why a thread is stopped:

- *The CPU*: will be interested in issuing instructions from some other live thread. In a simple pipelined CPU, that may involve discarding some instructions from the stopped thread, if they’ve already entered the main pipeline.
- *The OS*: may be interested in taking control when a thread is blocked for a long time - the TC could be in principle given another thread which might be able to make more progress. The OS overhead in changing the TC to another thread - really the same job as a thread-switch on a conventional CPU - is likely to be more than 100 instructions so the OS should only do this when the thread is likely to remain stopped for many hundreds of cycles.

  But it’s important that the OS has the power to take a blocked thread and detach it from its TC cleanly, so it can be restarted. That motivates some of the key features of the architecture, including the details of *Gating Storage*, see Section 2.7.1, "Gating storage".

2.5 Thread-scheduling decisions and the policy manager

The MIPS MT architecture is agnostic about thread scheduling. The immediate choice of which thread to run next is made inside the core; in the absence of any directions to the contrary, this choice is required to be fair to TCs in the long run.

However, in MIPS Technologies cores we envisage a rather dumb in-core scheduler given long-term hints by a *Policy Manager (PM)* which, living outside the core, may be customized for specific applications.

In particular the *TCSchedule* and *VPESchedule* registers (if implemented at all) will typically be inside the policy manager block; so what they do is strictly implementation-dependent.

The way the in-core scheduler in the 1004K core works is described in Section 3.2.1, "How the Dispatch Scheduler Works", and the choice of policy managers available from MIPS Technologies is in Section 3.2.3, "MIPS Policy managers included with the 1004K‘ core family".

2.6 Multithreading, exceptions and interrupts

An exception in a single-threaded MIPS architecture CPU is usually quite disruptive in the pipeline, and is commonly implemented by discarding a lot of execution state (pipelines get flushed and instructions discarded). An exception on a MIPS MT machine happens within a thread context - and other threads (at least those on separate VPEs) expect to continue undisturbed. So you'd expect there to be some difficulties when we redefine exceptions on a multithreading machine.

There are two types of exceptions:

- Interrupts are "asynchronous" - they happen for reasons unconnected with any particular instruction and are discussed in Section 2.6.1, "Multithreading and interrupts" below.

- Synchronous exceptions, associated with a particular instruction. That's what we'll look at first.

Bear in mind that an OS is a program (a set of threads, in fact). It's not characterized by the TC which happens to execute some part of it. The OS' exception handlers are each separate threads in their own right, in the meaning given by our definition of *Thread*.

Synchronous exception handlers are run by the TC whose instruction caused the exception. The TC immediately ceases work on its thread and starts fetching instructions from the appropriate exception handler.

The MIPS MT ASE requires that once a TC enters exception state, all the other TCs within the same VPE are suspended. None of the other TC's instructions may be executed until the VPE's *Status[EXL]* bit is cleared[5] by the exception handler. The exception handler (a new thread, remember) runs with kernel privileges and has access to all the defined CP0 registers, Because only one TC can be in exception state, the exception-related CP0 registers need only be replicated per-VPE.

In your MIPS MT system an exception not only causes a hiccup to the thread which takes it, but also suspends unrelated threads in the same VPE. If your application needs to maximize concurrency, you should consider minimizing exceptions - you may be able to use a thread blocked on an ITC access or **`yield`** condition instead. And, of course, arrange that exception handlers (as soon as they can) save the state necessary that they can drop back out of exception mode.

2.6.1 Multithreading and interrupts

In the MIPS architecture interrupt management is by CP0 registers (in particular, *Cause* and *Status*). Those registers are replicated per-VPE, not per-TC; so interrupt masking and steering is managed per-VPE. Even interrupt "wiring" into the core is per-VPE.

Each interrupt input may be connected to just one VPE or to all of them: ask your hardware engineer. In some systems you may be able to redirect interrupts (outside the CPU) under software control. If you connect and unmask an interrupt on multiple VPEs, any number of them may take the interrupt exception - you probably don't want that to happen, so either don't connect or don't enable some of them...

The interrupt exception may be taken by any available TC associated with the VPE.

[5] That may seem somewhat restrictive, but is necessary: critical exception handling state in the CP0 registers is not replicated per-TC, only per-VPE.
And it's not so bad as it looks, because it's already good practice to minimize the amount of code which runs with *Status[EXL]* set.

The MIPS architecture already provides multiple ways to refuse an interrupt exception: an interrupt to any thread from this VPE can be prevented by exception mode, a global interrupt-enable flag which may be zero, and by per-interrupt mask bits: that is by *Status[EXL]*, *Status[IE]* and *Status[IM]*[6]. The MIPS MT architecture adds yet another reason not to take an interrupt. You can now set a new per-TC CP0 register field *TCStatus[IXMT]* to make the TC *Interrupt exempt*. That will prevent the particular TC from being used for an interrupt exception. It's most obvious use is to permit some TC to run a thread which benefits from living in an interrupt-free universe.

2.7 Multithreading, non-blocking loads and stores, and gating storage

Most modern MIPS architecture cores implement non-blocking loads: that is, the core does not simply stop and wait for the load data to arrive. Instead, the register target of the load is marked and computation continues. If the data arrives before the program tries to use it, the data is sent directly to the register. But if some other instruction wants to read the register before the data arrives, the "consuming" instruction waits.

That means that a thread in a MIPS MT machine which does a “slow” load stops on the consuming instruction. When that happens the TC is still holding resources (e.g. the "fill buffer" in the CPU’s bus interface unit which remembers the load, waits for the data, and associates it with the register).

If you are using long-delayed loads as a means of synchronizing your application, non-blocking loads are unwelcome: it would be preferable for the thread to stop on the load itself.   So we provide a way to do that: memory locations used for synchronization can be mapped as *gating storage*.

2.7.1 Gating storage

The MIPS MT ASE provides for a kind of storage location whose behavior is adapted to loads which might be quite long-delayed, and which you may want to use for intentional thread synchronization. Such a location is called *Gating Storage*. A thread loading from a memory region marked as “gating” will block on the load itself. This is not the standard way of doing things: a thread which reads from a normal location which is slow to respond would run on until it attempted to use the data (that’s a “non-blocking load”).

It turns out to be useful to generalize this to writes as well as reads: even stores to gating storage locations block until the core gets an indication that everything went OK.

If a thread is blocked on a gated storage access and the OS decides that one of its valuable TCs has been hanging around too long, then the OS can take action. If the OS writes a 1 to the TC’s *TCHalt* register any gating storage access will be aborted, with the *TCRestart* address set to re-execute the load/store. Once the TC is safely halted, the OS can decide to use the TC for something else. When the thread is eventually scheduled again, the load instruction will be re-executed. Meanwhile the CPU hardware can forget about it.

The core interface provided for gating storage locations also permits external logic to abort an uncompleted load or store. Perhaps it’s better to describe this as “complete the operation with an exceptional condition”. The thread doing the access gets an exception, with the restart address set so the load/store will be retried after the exception. The gating storage exception is synchronous, and you’re guaranteed that the restart location captured in *EPC* will point to the load/store (or a preceding branch, if the load/store is in a branch delay slot). The exception can only happen if the thread is still waiting for the load/store, and the thread isn’t otherwise prevented from running.

If required an OS can take control of all GS load/stores; set *VPEControl[GSI]* and all GS accesses trigger an exception.

[6] This list is not comprehensive.

Out on the gating storage interface, no external party can see whether a TC is waiting or not. All GS transactions involve delivering something which waits around until the other side responds (some software books call this kind of synchronization a *rendezvous*).

Gated storage provides the opportunity to provide *ITC* locations - a form of what some of you may have read about before as “full/empty storage”. The ITC implementation which is optional in the 1004K core is described in Section 3.3, "Inter-thread communication storage (ITC)" below.

2.8 MIPS® Multithreading ASE - new instructions

There are very few extra instructions:

- **`fork rd,rs,rt`**: fires up a thread on a free TC (if available, see below). *rs* points to the instruction where the new thread is to start, and the new thread’s *rd* register gets the value from the existing thread’s *rt*.

  Some vital per-TC state is copied from the parent:

  - *TCStatus[TKSU]:* whether you’re in kernel or user mode — the same as *Status[KSU]*);
  - *TCStatus[TASID]*: what address space you’re part of — the same as *EntryHi[ASID]*;
  - *UserLocal*: some kind of kernel-maintained thread ID, see more in Section C.4.2 “The UserLocal register”.

  When the thread has finished its job it should use **`yield $0`** to free up the TC again.

  `fork`/**`yield`** are the only MIPS MT instructions usable in user mode (they’re also highly original, and are likely not to be extensively used in early MIPS MT architecture applications using substantial OS layers - they might be hidden inside the OS, but you won’t see them for a while in Linux user code).

  `fork` will only select a TC which is both "free" (*TCStatus[A]* is currently zero) and which is specifically marked as usable by fork because *TCStatus[DA]* is set.

  `fork` may fail if a suitable TC isn’t waiting at the “taxi-rank”. In that case you get an exception (“Thread Overflow”) which an OS may catch and fix up before restarting the application; that way the application remains unaware of the problem. This provides the illusion of an indefinite supply of TCs, in the same way that a virtual memory system provides an indefinite supply of memory - you’ll hear this described as that “**`fork`** has been virtualized” or made *Virtualizable*.

  `yield $0` has a matching "Thread Underflow" exception, which occurs when you’re about to reach a situation where all for-hire TCs are parked (because then the system might stop forever, with no threads running the code which might make another thread run...).

  There’s a lot more to say about yield, see the bullet below and Section 2.8.1, "Yield, Yield Qualifiers and threads waiting for hardware events".

- **`mftr rd,trno,u,sel,h`** and **`mttr rt,trno,u,sel,h`**: are privileged (CP0) instructions ("move to/ move from thread register") which provide read/write access to another TC’s registers.

The other TC is identified by *VPEControl[TargTC]*. The **trno,u,sel** fields identify which register of that TC you are accessing. Their encoding is complicated: we’ll present details in Table 2.1 below, but here’s a quick summary:

- When **u**==0, **trno** is a CP0 register and **sel** is the auxiliary 3-bit "select" field found in **mtc0**/**mfc0**;
- When **u**==1 and **sel**==0, **trno** is a general purpose register;
- When **u**==1 and **sel**>0, you get to access more exotic registers, as detailed in Table 2.1 below.

The **h** value should be specified as 1 when you’re obtaining the high half of a register which is double the size of a GPR. In other cases, omit it. However, this argument is *not* required for the multiply unit accumulators, where the low and high half have separate **trno** register numbers.

That’s fairly confusing, and the details are presented again in Table 2.1

.

Table 2.1 MTTR/MFTR - "U" and "SEL" values

u	sel	**trno**	***Other TC’s register type***
0	0-7	0-31	CP0 registers.
1	0	0-31	General-purpose integer registers
1	1	0 1	Multiply unit *lo* and *hi* respectively.
		4 5	Low and high half (respectively) of DSP accumulator 1
		8 9	Low and high half (respectively) of DSP accumulator 2
		12 13	Low and high half (respectively) of DSP accumulator 3
		16	*DSPControl* register.
1	2	0-31	Floating point (CP1) registers
1	3	0-31	Floating point control registers, as usually accessible with **cfc1**/ **ctc1**.
1 1	4 5	0-31 0-31	Co-processor 2 data and control register sets, respectively. Implementations are free to define large CP2 register sets; the MT ASE provides an extra 5-bit "rx" field to provide more bits for selecting the CP2 register, but the MT ASE does not define a standard assembler syntax to generate it.

The hardware does nothing, inherently, to make sure that register changes as a result of other-thread activity are seen tidily; unless you are really sure that the other thread is currently leaving the register alone, it’s safer to ensure that the other TC is halted (shut down for maintenance) before **mftr**/**mttr** will work reliably. One example where you really want to halt the other TC is when it is blocked on a gating storage access - a **mftr**/**mttr** accessing the result register for that will also block until that is resolved.

When disassembling binary code it is painful to have to hand-decode the **trno,u,sel,h** fields, so tool providers are recommended to support the alternative “idioms” described in Table 2.3 below, which are probably more memorable than binary numbers. Most tools will be symmetric, so you will be able to write the idioms too: but that doesn’t necessarily mean you *should* write code with them. You will, I hope, use meaningfully-named C preprocessor constants for all the various fields in your assembly source code, so it may be kinder on those who come after you if you expect them to remember just the **mttr**/**mftr** mnemonics.

Note that access to the registers of a TC affiliated to a different VPE is available only when *VPEConf0[MVP]* is set - it's often used as a safety-catch. In some environments (where you're not meant to be able to get at the other VPE's state) you'll find you can't set *VPEConf0[MVP]*.

If you attempt to read a register number which is not valid on your CPU, you will get an all-ones (-1) value back.

- **`dmt`**: suspend all other threads affiliated to the same VPE.

  Under the hood this atomically clears the *VPEControl[TE]* bit, returning the original value of *VPEControl* to an optional register argument[7]; so it is convenient to bracket a piece of code which needs to be single-threaded within the VPE by:

  ```
  dmt rt
  ehb    # need hazard barrier to be sure it took effect
  ...    # guaranteed to be the only live TC in this VPE
  mtc0 rt, VPEControl
  ```

  The "hazard barrier" should always be used when you change some CP0 condition and need to know it's taken effect when you run a subsequent instruction - see Section 7.1, "Hazard barrier instructions".

  OS code which updates registers and resources which are only replicated per-VPE will typically need this kind of protection, unless already multithreading-protected by something higher-level.

  The **`emt`** instruction atomically sets the *VPEControl[TE]* bit and returns the old value. It is relatively rarely used; it's more robust to replace the whole original value of *VPEControl* with an **`mtc0`**, because then things still work if you inadvertently nest one single-threaded block within another.

- **`dvpe`**: suspend all other threads, even those in other VPEs. In many systems VPE independence is much prized, and then this instruction is likely to be restricted to initialization software. Under the hood it clears the *MVPControl[EVP]* bit, returning the old value. Again, there's an **`evpe`** instruction, but a single-threaded block is better terminated by restoring the whole *MVPControl* register with an **`mtc0`**.

- **`yield rd,rs`**: a multi-purpose instruction, whose action depends on the value in *rs*. If and when it returns, *rd* is set to a bit-vector which shows the active inputs to **`yield`** - at least those enabled by the *YQMask* register. More in the section below.

  So:

  - When *rs* == 0: (also discussed under the bullet called "fork" at the start of Section 2.8, "MIPS® Multithreading ASE - new instructions") terminate the thread and clear the *TCStatus[A]* bit, permitting re-allocation to another purpose by **`fork`**. If this was the only live TC with *TCStatus[DA]* set (that is, the last TC in the **`fork`** pool), you get a "thread underflow" exception.

  - When *rs* == -1: polite pause while other threads get a chance to run. To be more precise, the thread will be stopped briefly while the yield indication is sent out to an external scheduling policy manager, if fitted (see Section 2.5, "Thread-scheduling decisions and the policy manager".) Such a policy manager may respond, in particular, to changes communicated by writing the *TCSchedule* and/or *VPESchedule* registers.

    After this sort of **`yield`** this thread will not run again for long enough that the policy manager has time to respond. But the thread hasn't been stopped and will normally run again soon, at the priority newly determined by the policy manager.

[7] They fit in with the encoding already used for atomic update of a CP0 register by the disable-interrupts instruction **`di`** etc.

- When *rs* == -2: has no scheduling effect, purely done for the value delivered to *rd*. And a **`yield -2`** never produces a “yield scheduler” exception.
- when *rs* > 0: waits for one or more of a set of signals to be asserted; from up to 31 signals available on your CPU, it is sensitive only to those selected by a “1” bit in the *rs* value. That’s complicated, see Section 2.8.1, "Yield, Yield Qualifiers and threads waiting for hardware events" below.

  But in particular, if the *rs* value includes a bit which is *not* set in *YQMask*, you get an "invalid qualifier" exception.

Software can ensure that *any* **`yield`** which would deschedule a thread (or any **`yield -1`** whose return status would be zero) produces a "yield scheduler" exception. A secure OS might do that because it wants to “scrub” the TC’s registers of any application data before the TC is returned to the free pool. To achieve this effect set *VPEControl[YSI]* (the "did any work" test depends on *TCStatus[DT]*.)

A **`yield`** instruction must not be in a branch delay slot.

2.8.1 Yield, Yield Qualifiers and threads waiting for hardware events

When the *rs* argument of the **`yield rs`** instruction is positive, the thread waits for a hardware condition; the thread will wait until the bitwise-and of *rs* and the hardware signal vector is non-zero. This is a cheap and efficient mechanism to get a thread to wait on the state of some input signal.

Cores in the 1004K family may have up to 16 external hardware signals attached. Because the **`yield`** instruction is available to user (low-privilege) software, you might not want it to have sight of all your hardware signals. The CP0 register *YQMask* is a bit-field where a “1” bit marks an incoming signal as accessible to the **`yield`** instruction.

In any OS running more threads than TCs you might want to reclaim a TC blocked on such a **`yield`**. If you need to do that while continuing to monitor the condition, then you’ll probably want your system integrator to ensure that the yield condition is also available as an interrupt, so you can get the OS’ attention when the condition happens.

The OS can zero-out corresponding bits 0-15 of *YQMask* to prevent them being used - a **`yield rs`** which attempts to use one of those bits will result in an exception.

In the two-operand form **`yield rd,rs`** the **`rd`** register gets a result, which is a bit-map with a 1 for every active yield input which is enabled by *YQMask* (bits which are zeroed in *YQMask* may return any value, don’t rely on them). The single-register form **`yield rs`** is really **`yield $0,rs`**.

2.8.2 All MT instructions in alphabetical order

That’s in Table 2.2 - but not quite all. There are a large number of convenience mnemonics (“assembler idioms”) which are not separate instructions, but which map onto some variant of the access-other-TCs-register instructions **`mttr`** and **`mftr`**. Rather than fill the table with these, we’ve consigned them to Table 2.3 below. If you’re looking up an unfamiliar instruction, please look in both tables.

Table 2.2 MT instruction summary in alphabetical order

Instruction	***Brief Description***
dmt rt	Clear *VPEControl[TE]*, which suspends execution of all other TCs affiliated to the same VPE. The *rt* register receives the original value of *VPEControl*; if you don't specify *rt* it defaults to *$0*.
dvpe rt	Disable all multithreading, including any other TCs affiliated to other VPEs, leaving this thread running alone. Implemented as an atomic clear of the *MVPControl[EVP]* bit. If you specify a register *rt* it receives the previous contents of the *MVPControl* register.
emt	The "enable" pairs of **`dmt`**/**`dvpe`** respectively. You may not need these instructions: when you've finished a section of code which must be single-threaded in some sense, it may be preferable to restore the whole *VPEControl*/*MVPControl* register from the value you got back when you ran the disable instruction, as suggested in the description of dmt in the running text above.
evpe	
fork rd,rs,rt	Find a TC and activate it, so it starts at *rs*. The new thread's *rd* register will be set to the value provided in *rt*. Lots more details above.
mftr rd,trno,u,sel,h	"Move from thread register" and "Move to thread register" - get/set the value of a register belonging to some other TC, using the general-purpose register *rd* as a sink, or *rt* as a source. For a per-VPE register, you will access the VPE affiliated to the target register - so to access a VPE first set up a TC affiliated to it. The other TC is identified by *VPEControl[TargTC]*, and the register you're accessing is selected by all of *trno*, *u* and *sel* - as described above or in Section 2.1, "MTTR/MFTR - "U" and "SEL" values".
mttr rt,trno,u,sel,h	
yield rd,rs	A multi-purpose instruction, whose action depends on the value in *rs*. When *rs*==0, it terminates the thread and makes the TC available for a subsequent **`fork`**. When *rs*==-1, pauses while other threads run and any scheduling policy change filters through. **`yield`** with *rs*==-2 is just done to poll yield inputs. When *rs*>0, you wait for one of the yield input signals, but only one for which there's a corresponding bit set in *rs*.

Table 2.3 MTTR/MFTR "assembler idioms" in alphabetical order

Write as	*Equivalent*	*Description*
`cftc1 rd,ft`	**`mftr rd,ft,1,3`**	Get data from/send data to another TC's floating-point (coprocessor 1, CP1) control register *ft*.
`cttc1 rt,ft`	**`mttr rd,ft,1,3`**	
`mftc0 rd,rt`	**`mftr rd,tc0r,0`**	Read other TC's CP0 register.
`mftc0 rd,rt,sel`	**`mftr rd,tc0r,0,sel`**	
`mftc1 rd,ft`	**`mftr rd,ft,1,2,0`**	Read low 32 bits from other TC's floating point data register.
`mftdsp rd`	**`mftr rd,16,1,1`**	Read other thread's *DSPControl* register.
`mftgpr rd,rt`	**`mftr rd,rt,1,0`**	Read other thread's general purpose register *rt*.
`mfthc1 rd,ft`	**`mftr rd,ft,1,2,1`**	Read high 32 bits from other TC's floating point data register.
`mfthi rd`	**`mftr rd,1,1,1`**	Read the other TC's *hi* multiply/divide unit register, which is the same as the first of...
`mfthi rd,ac0`	**`mftr rd,1,1,1`**	Read the high half of one of the other TC's *ac0-3* DSP accumulators.
`mfthi rd,ac1`	**`mftr rd,5,1,1`**	
`mfthi rd,ac2`	**`mftr rd,9,1,1`**	
`mfthi rd,ac3`	**`mftr rd,13,1,1`**	

Table 2.3 MTTR/MFTR "assembler idioms" in alphabetical order

Write as	*Equivalent*	*Description*
mftlo rd	**mftr rd,0,1,1**	Read the other TC's *lo* multiply/divide unit register, which is the same as the zeroth of...
mftlo rd,ac0	**mftr rd,0,1,1**	Read the low half of the other TC's *ac0-3* DSP accumulators.
mftlo rd,ac1	**mftr rd,4,1,1**	
mftlo rd,ac2	**mftr rd,8,1,1**	
mftlo rd,ac3	**mftr rd,12,1,1**	
mttc0 rt,rd	**mttr rt,tc0r,0**	Write other TC's CP0 register.
mttc0 rt,rd,sel	**mttr rt,tc0r,0,sel**	
mttc1 rt,fd	**mttr rt,fd,1,2,0**	Write data from *rt* to the low half of the other TC's floating point register *fd*.
mttdsp rt	**mttr rt,16,1,1**	Write to the other TC's *DSPControl* register.
mttgpr rt,rd	**mttr rt,rd,1,0**	Write to the other TC's general purpose register *rd*.
mtthc1 rt,fd	**mttr rt,fd,1,2,1**	Write data from *rt* to the high half of the other TC's floating point register *fd*.
mtthi rt	**mttr rt,1,1,1**	Write to the other TC's *hi* multiply unit register, which is the same as the zeroth of...
mtthi rt,ac0	**mttr rt,1,1,1**	Write to high part of the other TC's *ac0-3* accumulator.
mtthi rt,ac1	**mttr rt,5,1,1**	
mtthi rt,ac2	**mttr rt,9,1,1**	
mtthi rt,ac3	**mttr rt,13,1,1**	
mttlo rt	**mttr rt,0,1,1**	Write to the other TC's *lo* multiply unit register, which is the same as the zeroth of...
mttlo rt,ac0	**mttr rt,0,1,1**	Write to low part of the other TC's *ac0-3* accumulator.
mttlo rt,ac1	**mttr rt,4,1,1**	
mttlo rt,ac2	**mttr rt,8,1,1**	
mttlo rt,ac3	**mttr rt,12,1,1**	

2.9 Multithreading ASE - CP0 (privileged) registers

All the CP0 registers new to or affected by the MT ASE are in Table 2.4.

Table 2.4 CP0 registers required by MIPS® MT ASE

Register	*Description*
Per-TC registers	
TCStatus	Per-TC run-time control/status fields. Includes alternate views of per-TC fields in old CP0 registers.
TCBind	VPE affiliation and own TC number of this TC.
TCHalt	Per-TC - write one to halt the TC for maintenance, zero to let it run again. No further description needed.
TCRestart	Per-TC - address of instruction the TC will run next. Unambiguous only when the TC is halted. Writing *TCRestart* (to control where the TC executes from next time it is made live) has side-effects; in particular it clears the link bit which associates a load-linked/store-conditional pair, see Section 3.5, "Synchronization: "ll" and "sc" instructions implementation".
TCContext	per-TC 32-bit read-write scratch register for OS use, no hardware-interpreted fields.
Per-VPE registers	
VPEControl	Per-VPE - status and control fields for exception and `mftr`/`mttr` instruction support.
VPEConf0-1	read-only status of VPE setup
YQMask	bitfield where “1” bits define valid select bits for a `yield` instruction - see Section 2.8.1, "Yield, Yield Qualifiers and threads waiting for hardware events".
VPEopt	Can be used to mark any cache "way" (a quarter) of the primary I- and D-caches as inaccessible to the owning VPE, to keep it clear for the other one.
SRSConf0-4	Fixes which TC’s GPRs are recycled as shadow register sets.
Whole-CPU control and availability of MT resources	
MVPControl	writable register to determine how multiprocessing facilities work.
MVPConf0-1	read-only summary of CPU MT resources
Software hints and controls on thread scheduling	
TCSchedule	Per-TC, writable register to influence thread scheduling. It’s not really part of the core, and the description is for our sample thread scheduling policy manager.
VPESchedule	Per-VPE, writable register to influence scheduling
TCScheFBack	Optional read-only register providing statistical information about thread scheduling.
Software Thread ID register (not just MT CPUs)	
UserLocal	This plain 32-bit register is replicated per TC. It’s not hardware-interpreted at all, but its value is readable by a user-space program using `rdhwr $29`. It’s intended for use by OS kernels which provide some kind of thread ID for user-space software, typically a thread library of some kind. Its value is inherited by the child TC of a successful `fork`. More information in Section C.4.2 “The UserLocal register”.
New fields in old registers	
EBase[CPUNum]	Identity of running VPE within CPU
Config3[MT]	set if this CPU implements the MIPS MT ASE.
Debug[OffLine]	a per-TC bit which a debugger can set to quiesce a TC while it debugs another thread (but without affecting any non-debug state).

2.9.1 What CP0 registers are per-TC, per-VPE and per-CPU?

At first sight the CP0 register map looks like quite a chaotic mixture of fields replicated per-TC, per-VPE or not replicated at all. But in fact the rules are fairly simple, and there are only a few special cases:

- All registers called “TCxx” and *UserLocal* are per-TC.

- All other CP0 registers (not called "TCxx") are per-VPE except for:
  - *MVPControl*, *MVPConf0-1* are not replicated, there's just one set on a CPU.
  - The performance counter count and control registers are per-TC.
  - A handful of fields in pre-MT MIPS32-standard registers are replicated per-TC: they include those which are found in *TCStatus* as fields called "Txx", plus the debugger controls *Debug[OffLine]* (a "thread halt" control for debuggers) and *Debug[SSt]* (single-step).

Like all other CP0 registers, many fields are not initialized by hardware when the CPU is reset. Unless you are confident that random contents in some particular register are safe, it's your responsibility to write registers to sensible values.

2.9.2 VPEControl

Figure 2.1 Fields in the VPEControl register

	31 22	21	20	19	18 16	15	14 8	7 0
`VPEControl`	0	YSI	GSI	0	EXCPT	TE	0	TargTC

In *VPEControl*:

YSI, GSI: "intercept" bits for yield and *Gating Storage* operations. By setting one or both of these bits 1, an OS can arrange to be notified (by an exception) if any thread would otherwise become blocked by a **`yield`** instruction, or on an access to gating storage. The exception will only happen if the TC's *TCStatus[DT]* bit is set, that is if the TC has run an instruction since it was last deallocated.

YSI affects any **`yield`** instruction which would block; but a **`yield`** which tests for a condition which is already true, or a **`yield -2`** will not be affected (a **`yield -2`** is just a poll - see the bullet on "yield")

GSI affects any gating storage access which will block the thread[8].

EXCPT: encodes the cause of the last thread exception. This refines the information returned by *Cause[ExcCode]* - we don't have enough reserved values to encode all the thread exceptions separately. Like the old cause register field, *VPEControl[EXCPT]* is only valid so long as the TC remains in exception mode (recall that in exception mode only one TC within the VPE may run).

The possible values are:

Table 2.5 Thread exception codes in VPEControl[EXCPT]

0	Thread underflow on **`yield`**.
1	Thread overflow on **`fork`**.
2	Bad qualifier fed to **`yield`**.
3	Exception on gating storage operation
4	**`yield`** which would have blocked run while *VPEControl[YSI]* is set to 1.
5	Gating storage access which would have blocked attempted while *VPEControl[GSI]* is set to 1.

[8] Gating storage operations are uncached, and may be slow; but *GSI* won't lead to an exception because the access is slow, only if the gating storage interface is told that the operation is blocked.

TE: "enable multithreading" - when clear, only one TC attached to the VPE is allowed to issue instructions. You normally set/unset this using the **`dmt`**/**`emt`** instructions.

TargTC: selects the TC number of the "other thread context" in the **`mttr`**/**`mftr`** instructions - TCs are numbered from 0 upward. There isn’t room to encode the TC number in the instructions. Note that since the whole of *VPEControl* is a per-VPE register, TC-multithreaded software will need some kind of lock (perhaps **`dmt`**/**`emt`** brackets) around any code which uses **`mttr`**/**`mftr`**.

2.9.3 TCRestart, TCHalt and TCContext

Three registers without internal fields:

- *TCRestart* holds the thread’s “PC” - more accurately, when the TC is halted it holds the instruction address where the TC will restart. If a TC is to retry an instruction in a delay slot, *TCRestart* will point to the branch but the *TCStatus[TDS]* flag will be set.

- *TCHalt*: just write a 1 to the register, and the TC will halt and will be safe to inspect and reprogram. Write a zero to let it run again. *TCHalt* is for the use of MT-aware OS code.

- *TCContext* is a pure 32-bit software register, without any hardware effect. OS software finds it useful to have a per-TC register where it can write an ID or key pointer which identifies the thread.

2.9.4 TCStatus

Figure 2.2 Fields in the TCStatus register

31 28	27	26 25	24 23	22	21	20	19 18	17	16	15	14	13	12 11	10	9 8	7 0
TCU3-0	TMX	0	RNST	0	TDS	DT	0	TCEE	0	DA	0	A	TKSU	IXMT	0	TASID

TCU3-0, TMX, TCEE, TKSU, TASID: These fields - most of those starting with a "*T*", in fact - provide convenient alternate views of some fields in legacy CP0 registers. They are fields which, with MIPS MT, need to be replicated per-TC. This is valuable because code can change them without the difficulty of doing a non-atomic read-modify-write sequence on one of the legacy registers (which would mean having to read-write many fields which are shared with any other TCs in the VPE.)

The fields listed are views of the *Status[CU0-3]*, *Status[MX]*, *Status[CEE]*, *Status[KSU]* and *EntryHi[ASID]* fields respectively, and to get a complete view of what any of them do you are recommended to look at the notes on each of those CP0 registers.

We’ll deal with these alternate-view fields first:

TCU3-0: set a bit to enable this TC to run instructions for the corresponding co-processor. There are four bits but only two feasible co-processors: CP1 is the floating point unit (if fitted) and CP2 is available for custom use. CP3 is not available on cores in the 1004K family, so *TCStatus[TCU3]* always reads zero.

The floating-point unit available as coprocessor 1 for 1004K family cores has only one set of registers, so it may only be used by one TC. It is the OS’ responsibility to make sure that’s done. Custom or future coprocessors may replicate all their state per-TC (so they may be freely multithreaded) or provide fewer, perhaps just one, register set.

TMX: another view of *Status[MX]*, which is the enable bit for the instructions in the MIPS DSP ASE (in theory it plays the same role for the older but less-used MDMX, but that will never be found in a 1004K family core.) It’s visible here because it’s a per-TC field.

TCEE: another view of the per-TC *Status[CEE]* enable bit implemented by a "CorExtend" (user-defined) instruction block which needs it, usually because it includes some registers which it may need a kernel to save across exceptions and context switches.

TKSU: a convenient view of the per-TC *Status[KSU]* bits which determines the privilege state of the CPU.

TASID: a convenient alternative view of the per-TC *EntryHi[ASID]* field.

RNST: (read-only) status, which can be used to find out why a blocked TC is blocked, with values meaning:

0	Not blocked.
1	Asleep after a **`wait`**.
2	Blocked on **`yield`** (that is, waiting for one or more of the *Yield Qualifier* signals to activate).
3	Waiting for gating storage load/store to complete.

TDS: qualifies the per-TC restart address *TCRestart*, indicating that the thread is stopped in a branch delay slot (in which case *TCRestart* will point to the branch.) An analogue of *Cause[BD]*.

DT: "dirty" bit - set whenever the thread being run by the TC makes progress. More precisely, set when any of this TC’s instructions completes (though instructions in exception, error-exception or debug mode are not counted); it is also set if the TC is successfully started as a result of a **`fork`**.

This is for the use of an OS overseeing applications forking at user level; it can inspect its free-TC pool and discover which ones have done any work since last time it looked. This may be important, because a TC which has done work for one application[9] might have some of that application’s data in its registers, and cannot be automatically allocated to a different application for fear of leaking data (applications are not supposed to see one another’s data). The OS must make sure it scrubs all the TC’s registers before that happens; this bit is part of the mechanism which lets it find out when it needs to scrub.

DA: "dynamically allocatable" - when clear, this TC may not be allocated as a result of a **`fork`**. If as a result a **`fork`** can’t find a TC to use, it takes a "thread overflow" exception.

If the only dynamically allocatable and live TC executes a **`yield`** (which might lead to the silence of the grave), it takes a "thread underflow" exception instead.

A: "activated" (sometimes, "allocated"). Can’t run instructions without this bit, which is set by **`fork`** and cleared by **`yield $0`**.

IXMT: set 1 to prevent this TC from handling interrupts.

Summary TC status

The *TCHalt* and *TCStatus[A,DA]* fields interact as shown in Figure 2.6 and may be best understood together. Note that the EJTAG debug Debug[OffLine] bit, if set, overrides all of them and prevents the TC from executing its thread. *OffLine*, though, doesn’t affect whether a TC may be selected by a **`fork`**.

[9] In a Linux context "process" would be more precise than "application".

Table 2.6 TC summary state as expressed in per-TC register fields

Register bits			*What happens to TC*
`TCHalt`	`TCStatus`		
	`[A]`	`[DA]`	
`1`	`X`	`X`	TC is halted, fit for inspection and maintenance by software running on some other TC
`0`	`0`	`0`	TC is not running, nor may it be used by **`fork`**.
`0`	`0`	`1`	TC is "parked at the taxi-rank" ready to be used as a result of a **`fork`** instruction
`0`	`1`	`X`	TC is working through the instructions of some thread. Maybe it’s not currently live, but that will be for a thread-specific reason.

2.9.5 TCBind

Figure 2.3 Fields in the TCBind register

31 29	28 21	20 18	17	16 4	3 0
0	CurTC	0	TBE	0	CurVPE

CurTC: (read-only) returns the TC’s own identity - the ID of the TC which ran the **`mfc0`** instruction, or which is the target of a **`mftr`**.

The zero field which occupies bits 18-20 below *CurTC* is reserved by the architectural definition. It can save a mask operation when you need to use *CurTC* as an index into an array of 4- or 8-byte objects.

TBE: set by hardware when a transaction causing a bus error is identified as resulting from a load issued by this TC: see Section 5.5, "Bus error exception" for details. It remains set until explicitly written to zero.

CurVPE: the ID number of the VPE affiliation of this TC. You write this field to change a TC’s affiliation, but only when the "VPE configuration mode" safety-catch bit *MVPControl[VPC]* is set. In principle it’s possible for a thread to set its own TC’s affiliation, but that seems fraught with difficulty. This will most often be set by some supervisory thread using an **`mttr`** instruction).

2.9.6 MVPConf0-1 - read-only multithreading-specific configuration information

The *MVPConf0-1* registers present a read-only summary of the CPU’s multithreading resources.

Figure 2.4 Fields in the MVPConf0-1 registers

	31	30	29	28	27	26	25 16	15	14	13 10	9 8	7 0
`MVPConf0`	M	0	TLBS	GS	PCP	0	PTLBE	TCA	0	PVPE	0	PTC

	31	30	29 28	27 20	19 18	17 10	9 8	7 0
`MVPConf1`	C1M	C1F	0	PCX	0	PCP2	0	PCP1

MVPConf0[M]: a "continuation" bit - if zero, *MVPConf1* isn’t implemented, and acts as if it was all-zero.

MVPConf0[TLBS]: 1 if it’s possible to share the TLB. To do that you’d have to set *MVPControl[STLB]*, see Figure 2.5.

MVPConf0[GS]: reads 1 if the CPU is able to support *Gating Storage* as described in Section 2.7.1, "Gating storage".

MVPConf0[PCP]: read-only bit. Reads 1 if it's possible to deny access to one or more primary cache "ways" to each VPE. This feature must be enabled in *MVPControl[CPA]* and the way inhibition programmed in *VPEopt*, as described in Section 2.9.10, "VPEOpt register - reserve some cache "way" for use of one VPE".

MVPConf0[PTLBE]: the number of TLB slots which may be provided to different VPEs according to initialization software. Will read zero - even on a CPU with a sharable TLB - if the TLB configuration has no option other than shared or split in some fixed way.

MVPConf0[TCA]: reads 1 on 1004K, because it is possible to dynamically assign TCs to a VPE (by writing *TCBind[CurVPE]*.) Other MIPS MT implementations may not let you do that.

MVPConf0[PVPE,PTC]: how many separate VPEs/TCs respectively are available on this CPU (the field encodes "number of things minus one", so that zero means "one VPE" (or TC).

MVPConf1[C1M]: the floating point unit (co-processor 1) implements the old MDMX™ extension to the instruction set. This will always be zero on CPUs in the 1004K core family.

MVPConf1[C1F]: co-processor 1 implements 64-bit floating point instructions as defined in [MIPS64V2].

MVPConf1[PCX,PCP2,PCP1]: how many register set contexts are available for CorExtend™, co-processor 2 and co-processor 1 respectively.

2.9.7 MVPControl Register - CPU-wide VPE control

MVPControl is a read/write per-CPU control/status register.

Figure 2.5 Fields in the MVPControl register

	31	4	3	2	1	0
MVPControl	0		CPA	STLB	VPC	EVP

MVPControl[CPA]: set 1 to enable the per-VPE *VPEOpt* registers (see Figure 2.7 and notes) to prevent a VPE from being allocated new lines in one or more ways of the primary caches. Check *MVPConf0[PCP]* first, to see whether this feature is available.

MVPControl[STLB]: set to enable TLB sharing between the VPEs, see Section 4.3.4, "Sharing and not sharing the TLB".

MVPControl[VPC]: "configuration mode" - a heavy-duty safety catch. When this bit is set to "1", it becomes possible to write to configuration register fields which are read-only on conventional MIPS32-compliant CPUs.

This is obviously a fairly dangerous thing to do, and it's unlikely to be a good idea to change the configuration registers except when launching a VPE with software which believes it is re-initializing itself. In particular, make sure that no other VPE is running by executing a **dvpe; ehb** sequence - the **ehb** ("hazard barrier") instruction makes sure that subsequent instructions don't start until the **dvpe** has taken effect.

But with this bit set ("unsafe"), a MIPS MT CPU can be set up by MT-aware software to configure a VPE with its choice of CPU resources, then pass that VPE to legacy (non-MIPS-MT-aware) software with that choice of resources presented by the standard *ConfigNN* registers.

With this bit zero, the fields in the *ConfigNN* registers revert to read-only.

MVPControl is writable only if the "master VPE" safety catch *VPEConf0[MVP]* is set to 1.

MVPControl[EVP]: when clear, instructions will only be executed for the thread which was running when this bit was cleared - even TCs affiliated to other VPEs will not be run. This bit is usually manipulated with the **`dvpe`**/**`mtc0`** instructions.

2.9.8 VPEConf0-1 registers - initializable per VPE resource lists

These are per-VPE registers which are read to show what resources are available to software on the VPE. Some fields are writable, but that's only when the VPE-access safety catch *VPEConf0[MVP]* is already set. That means that setting your own *VPEConf0[MVP]* to zero is an irreversible abdication from inter-VPE power if other VPEs do the same.

Figure 2.6 Fields in the VPEConf0-1 registers

	31	30 29	28 21	20	19	18	17	16	15 2	1	0
VPEConf0	1	0	XTC	0	TCS	SCS	DCS	ICS	0	MVP	VPA

	31 28	27 20	19 18	17 10	9 8	7 0
VPEConf1	0	NCX	0	NCP2	0	NCP1

VPEConf0[XTC]: When only one TC in a VPE is running because the VPE is in exception mode or *VPEControl[TE]* is clear, this field identifies that one running TC. *XTC* can be written by **`mttr`** as part of cross-VPE initialization if you want to initialize a VPE so it starts with just one TC running alone. Anything might happen if you tried to write this field on a running VPE, so you're prevented from doing so - the field is not writable unless the target VPE's *VPEControl[VPA]* is zero.

Of course the initializing thread, running such a **`mttr`**, will need its own copy of *VPEConf0[MVP]* set to do cross-VPE access in the first place.

VPEConf0[TCS,SCS,DCS,ICS]: read-only bits which tell software which caches are shared with at least one other VPE. The separate bits are for tertiary, secondary, L1 D-cache and L1 I-cache respectively. There's no way for a 1004K core to be fitted with un-shared caches, so a 1004K core will have *DCS* and *ICS* set (and will have the other bits set if it has L2 or L3 cache).

VPEConf0[MVP]: "master virtual processor" - a safety catch bit, which must be set before software can touch registers in different VPEs (or in the TCs of different VPE affiliation).

It also controls write access to *MVPControl*.

VPEConf0[VPA]: Virtual Processor Activated. If zero, no TCs bound to this VPE will run.

VPEConf1[NCX,NCP2,NCP1]: number of CorExtend, coprocessor-2 and coprocessor-1/floating-point contexts available to this VPE. These fields are writable at configuration time to zero, one or the number of TCs affiliated to the VPE[10] and will be reflected in the VPE's view of *Config[UDI]* (for CorExtend) and *Config1[C2,FP]*. If a thread within the VPE is to run a legacy OS, you can use that to determine whether the legacy software sees UDI, CP2 and/or floating point capability.

2.9.9 YQMask register - enable yield "conditions"

YQMask is a bit-map where you write a "1" bit to make the corresponding yield condition usable for the **`yield mask`** instruction, as described in Section 2.8.1 "Yield, Yield Qualifiers and threads waiting for hardware events".

[10] If this field was set to "number of TCs" but the number of TCs affiliated to the VPE subsequently changes (it can happen) the field will be automatically updated.

2.9.10 VPEOpt register - reserve some cache "way" for use of one VPE

Two OS programs running on separate VPEs of a MIPS MT CPU progress independently of each other, and the thread scheduling rules usually make sure that each gets a fair proportion of the CPU’s attention. However, one unavoidable interaction is that threads on both VPEs are competing for the same cache resources.

The 1004K core’s primary caches are 4-way set associative, and this is usually enough to provide for the active “working set” of all loaded threads.

Occasionally a critical routine may need such a good response time that it is unacceptable for it to be dislodged from the cache by an unrelated thread. Where this affects a tiny piece of code, your best bet is probably to lock the code concerned into the cache, as described in Section 5.4.11, "Cache locking".

But if some legacy code consigned to the independent environment of one VPE is suffering because of competition for cache locations from an unrelated program on the other VPE, you may also have the choice of reserving some part of the cache for the use of a VPE: to check whether this facility is available on your core, test *MVPConf0[PCP]*.

This facility is large-scale, affecting a whole cache “way” - that’s 25% of a cache, removing one out of the four cache locations available to store any particular cache-line sized piece of memory. It’s implemented by getting a VPE to sacrifice the ability to load data into one or more cache ways, effectively reserving it for the other VPE. Once the data is loaded, either VPE can access it.

Caution: *You almost certainly shouldn’t do this*. This is a facility offered to dig systems out of a very particular kind of hole. Only use it after careful measurement has convinced you that you have a problem caused by competition for cache resources, and keep measuring to make sure you’re getting the effect you need.

Figure 2.7 Fields in the VPEOpt register

	31 12	11 8	7 4	3 0
VPEOpt	0	IWX3-0	0	DWX3-0

But once you’re sure: to enable this facility at all, you need to set *MVPControl[CPA]*. Then to renounce the ability to obtain new lines from one of the four cache ways in the I- or D-cache for anything you miss on in future, set the corresponding *IWXnn*/*DWXnn* bit in the *VPEOpt* register, as shown in Figure 2.7. Since this is done on a per-VPE basis, you could try to completely deny yourself use of some part of the cache — such an operation will fail, silently.

After CPU reset these fields are cleared to zero, so if you don’t need this facility, just ignore it.

2.9.11 Shadow register configuration SRSConf0-4

A TC’s registers can be borrowed and used as a “shadow set” for another TC. These registers control how this is done. It seemed simpler to combine their description with the rest of the shadow register system in Section 7.4, "Shadow registers".

2.9.12 Thread scheduling hints - TCSchedule, TCScheFBack, VPESchedule

The *TCSchedule*, *TCScheFBack*, and *VPESchedule* registers are inputs to wholly implementation-dependent logic, so their description is not in this chapter, but in Section 3.2, "Thread scheduling in the 1004K‘ core" below.

Chapter 3

How the 1004K™ core implements multi-threading

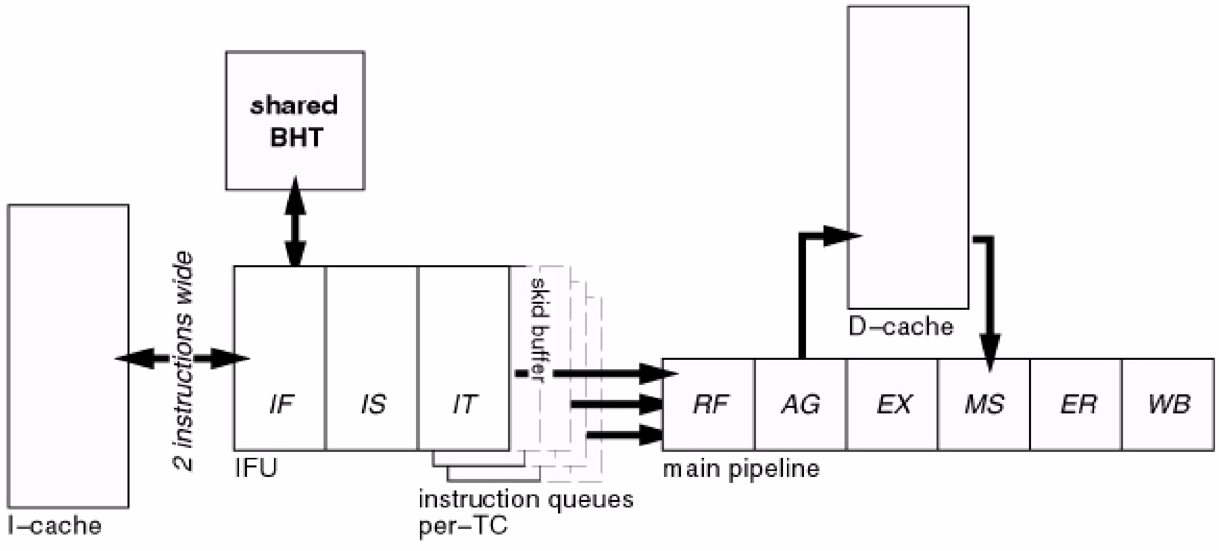
In this chapter:

- Section 3.1, "The 1004K™ core pipeline and multithreading" how it all runs.
- Section 3.2, "Thread scheduling in the 1004K‘ core"
- Section 3.3, "Inter-thread communication storage (ITC)"
- Section 3.4, "The 1004K™ core and interrupts"

3.1 The 1004K™ core pipeline and multithreading

The 1004K pipeline is shown in Figure 3-1. It inherits the 24K core’s basic pipeline.

Figure 3.1 The 1004K™ core pipeline



Notes on the pipeline diagram Figure 3.1:

The 1004K core issues one instruction per clock and executes instructions for a particular thread strictly in order. We’ll say an instruction is “fetched” when it’s read from the I-cache in the IF stage, it’s “issued” when it’s sent to the RF stage and “executed” when it emerges from the ER stage without causing an exception.

- *Instruction fetch unit*: The instruction fetch unit ("IFU") occupies the three first stages and is decoupled from the main pipeline.

Each TC in the processor has some of its own fetch unit state, and in particular each TC has a dedicated instruction queue which is kept filled whenever the TC is not stopped.

The IFU works a bit like a dog being taken for a walk. It rushes on ahead as long as the lead will stretch (the IFU, processing instructions two at a time, can rapidly get ahead). Even though you're in charge, your dog likes to go first - and so it is with the IFU. Like a dog, the IFU guesses where you want to go, strongly influenced by its observations of your habits. If you make an unexpected turn there is a brief hiatus while the dog comes back and gets up front again... but now we're anticipating the next bullet on "branch prediction". This kind of design is called a "decoupled" IFU.

Once instructions are issued from the IFU they are either completed in order or nullified (that is, essentially turned into a **`nop`** - such instructions continue to occupy a pipeline slot). When a thread stops for any reason (see Section 2.4, "When can't threads run?") any of its instructions which have entered the main pipeline after the "stopping" victim will be nullified; the fetch unit holds the last couple of instructions issued in its *Skid buffer*, so it isn't necessarily going to have to go back and fetch the instructions from the I-cache all over again.

Even going back to the skid buffer is an avoidable overhead if the thread which stopped was the last runnable one. The hardware may detect this condition and decide to stall the main pipeline with the post-blockage instructions still in it when it knows there are no other runnable threads (in the hardware documents this is called “single-threaded mode”).

- *Branch prediction*: The fetch unit has a couple of ways of predicting the branch target, allowing it to fill a TC's instruction buffer speculatively without waiting for the main pipeline to do calculations and report on branch conditions. It has:

  - *Target computation*: the fetch unit has logic which can compute the target of both PC-relative branches and long-displacement format **`jal`**/**`j`** instructions.

  - *A branch history table*: which is shared by all the TCs, is used to guess the direction of conditional branches. The table is indexed by the low address bits of the instruction's location, and keeps 2 bits of state for each slot. It's surprisingly effective, guessing right over 90% of the time. All branches (including the misnamed "branch likely" instructions) are treated the same.

  - *A return prediction stack*: a small stack on which the IFU pushes the return address of any subroutine call instruction. Subroutine return (i.e. **`jr ra`**) instructions pop the stack and guess that it delivers the correct target address.

    When multiple TCs are running, only one of them may use this stack. A TC gets to use the stack whenever all other TCs are blocked for relatively long-term reasons, and gets to keep it (even though conditions change and other TCs become unblocked) until some other TC qualifies.

  When the guess turns out to be wrong or the execution unit encounters an unpredictable computed branch the execution unit issues a *Redirect* and nullifies any of the TC's instructions in the pipe; the IFU has to discard all queued instructions for this TC, and start fetching again from the corrected address.

- *Main pipeline*: like the 24K core, the main pipeline is adjusted to provide something more than two clocks for accessing the L1 caches. It also shares the "skewed ALU" - load/store address calculation is done in the dedicated AG stage ahead of the EX stage where arithmetic and logical functions happen. The skewed ALU keeps the load-to-use delay down to just one clock.

  There's no such thing as a free lunch; the downside is that a load/store instruction whose address generation depends on the immediately preceding instruction will have to wait for one clock. Compilers probably find it easier to move the address calculation back one place in the instruction stream, rather than to find yet another useful instruction which can be moved between the load and use of the data. But code which follows a pointer chain is guaranteed to take at least three cycles per pointer.

3.1.1 Resource limits and consequences

The long pipeline, data interlocks, and the semi-autonomous IFU mean that the whole pipeline does not advance in lock-step as in the simplest MIPS architecture CPUs. Updates to internal states are not so easy to schedule at fixed times; instead they tend to wait in queues until a convenient moment. Most of the time, the convenient moment arrives quickly and there is no software-visible effect. But sometimes an unusual code sequence causes updates to be generated faster than they can be dealt with, the queue fills up and execution of the thread - perhaps the whole CPU pipeline - has to be stopped while the updates are done. The various relevant queues are discussed in Section 5.3.1 “Read/write ordering and cache/memory data queues in the 1004K‘ CPU”.

Outstanding actions which can fill up the queues include:

- *Cache refills in flight*: there can be four or eight D-cache refills (at build-time option), and two I-cache refills. In a single-threaded application you’re unlikely to reach this limit unless you are using prefetch or otherwise deliberately optimizing loops. If a series of prefetches use all available resources, the next unrelated load-miss will stall the pipeline.

  A hard-working multi-threading application might get there more often - hence the option to handle eight D-cache refills in flight.

- *Non-blocking loads to registers*: the 1004K core has enough resources to have one load outstanding on each TC. They’re used not just for non-blocking loads, but also for a TC blocked on gating storage. Compiled code is unlikely to reach this limit.

- *Lines evicted from the cache awaiting writeback (eight)*: the 1004K core’s ability to write data will in almost all circumstances exceed the bandwidth available to memory: this queue will absorb short bursts without delaying anything. A long enough burst of writes will eventually slow to memory speed.
  This same buffer is also used to return data to interventions (snoops) that hit in the local cache.

- *Register file write port*: only one instruction can write a register value in each clock. For instructions which execute down the main CPU pipe this is not in the least problematic: they arrive at their register-write stage one at a time in sequence. But some instructions with their own pipeline (multiply/divide operations, loads/stores, coprocessor operations), and any result from such an instruction which is delivered to a general-purpose register must wait for a slot in which the main-pipeline instruction doesn’t need to write a register. Typically, this happens very soon: but it depends on the instruction sequence.

More complicated interactions happen with some specialist operations, particularly the cache-management (**`cache`**) instruction: see Section 5.4.6 “L1 Cache instruction timing” for details.

3.1.2 Choosing what TC’s instruction to issue next

There’s a critical piece of logic called the *Dispatch Scheduler* running in the IS/IT pipeline stages. It’s job is to decide which TC’s instruction to issue next, and that’s the subject of the next section.

But if we look at the whole CPU, we see that the instructions in the main pipeline are not necessarily (nor even usually) all from the same TC. The hardware carries a TC number down the pipeline with each instruction, and that TC number is used to extend any register number defined by the instruction to read and write a register from the TC’s own set.

This is fine, so long as no threads block. When a thread blocks, the fetch unit gets to know about it and will not issue any more instructions for that TC; but by that time some more instructions for that TC are likely to be in the main pipeline. These instructions are now doomed, but must be allowed to continue through the pipeline: otherwise instructions from unrelated, unblocked TCs could not make progress[11]. The doomed instructions are marked as "nullified": they will cause no exception, no load or store, and no register write-back.

Meanwhile, the TC's own instruction queue is told of the blockage, and told where to restart after the thread is unblocked. The instruction at the head of the TC's instruction queue won't be the restart one (because one or more instructions were entered into the main pipeline and nullified). To avoid discarding the whole instruction queue and re-fetching all the instructions, the instruction queue includes a "skid buffer", which keeps a copy of a couple of instructions which have been issued but might still be nullified. So the TC which is stopped can back-up the skid buffer and wait to be running again.

3.2 Thread scheduling in the 1004K™ core

In any multithreading CPU you have somehow to determine which instruction to run next. Of course, this decision gets much easier if you have configured the core with a single TC. So much easier that a pipeline stage used for thread selection is completely bypassed in that configuration. Speaking of bypasses, feel free to bypass this section if you are working with such a core.

The logic which does this job in the 1004K core is called the *Dispatch Scheduler* ("DS"). On every cycle the DS selects an instruction from one of the per-TC instruction buffers and puts it into the main pipeline. Its decision is influenced by signals from the main pipeline but also by per-TC signals delivered from a piece of logic outside the core, the *Policy Manager (PM)*. The simplest PMs just tie some interface signals to fixed levels; there are others which just feed back some bit-fields from the *TCSchedule* and *VPESchedule* registers. Customers can use a MIPS-supplied PM or create their own - for more details see Section 3.2.3, "MIPS Policy managers included with the 1004K‘ core family".

Instructions are fetched at the front of the IFU: so how do we choose the TC for which we'll fetch a pair of instructions for this clock? That's fairly simple. Fetch will rotate through each running TC which has room on its instruction queue (though there are minor tweaks in the hardware so an empty queue gets attention quickly).

3.2.1 How the Dispatch Scheduler Works

The dispatch scheduler computes a priority for each TC. Where there are TCs with different priorities, it will do a cycle-by-cycle round-robin between the highest-priority TCs which are running.

The priority calculation includes the following bits, in something like this order:

- The MS bit represents "running": that means the priority mechanism automatically makes sure that we avoid selecting an instruction from a blocked TC (and we don't need any special purpose logic to do that).

- The priority may include a bit which is clear when the TC is waiting for cache miss or uncached-read data to become available, causing any other TC which has no pending load to get priority. If this feature is available, the *Config7[BTLM]* bit is writable - set it to 1 to enable the feature as described in Section C.4.5, "The Config7 register". This is likely to provide a small performance advantage to most general-purpose multithreading systems: in most code sequences the CPU will usually block waiting for data very soon after a load. A thread waiting for a load is therefore likely not to make much progress in the short term, and this feature will put resources

[11] If there is only one running thread (so nothing else can happen if the stopped thread is evicted from the pipeline) the whole 1004K pipeline may be stalled.

into an alternate runnable thread which is able to progress. Additionally, this can prevent a greedy thread from using up all of the load miss slots

But there can be special-purpose system (which issue critical loads far ahead of use, for example) where this is undesirable -- that’s why you have to set this explicitly.

- The priority includes the 2-bit per-TC priority which is supplied by the Policy Manager.

- The LS "priority" bits are for "round-robin" - again, the priority check is overloaded to implement the round-robin algorithm for otherwise-equal-priority TCs.

Along with the real TCs, the DS can have one or more *Relax* numbers (a bogus TC number different from all real TCs); when a "relax" number wins the arbitration no instruction is issued, and perhaps some energy is saved. This feature is controlled from the external policy manager (see below) and in particular the *VPESchedule* register.

3.2.2 The Policy Manager interface

The interface is hardware, really. But if you are programming a core equipped with a custom PM, you probably need to know something about the hardware interface.

The *TCSchedule* and *VPESchedule* registers (if implemented at all) are inside the policy manager and their values may influence its behavior in any way the designer thinks fit. The PMs supplied by MIPS to licensees are described in the next section.

The policy manager supplies the core with:

- A 2-bit "group number" for each TC, mapping each onto one of four "scheduling groups".

- A 2-bit priority for every group. A TC then gets a priority from its group, which influences the internal dispatch scheduler as to which TC’s instruction to schedule next.

- A per-TC "block" signal which when asserted freezes the TC completely. This is not used (that is, it’s hard-wired deasserted) in MIPS Technologies’ own PM designs.

- A set of "relax" signals corresponding to a bogus "relax" TC for each VPE; each has its own 2-bit priority and an enable.

The PM has access to many signals from the core. Per-TC information includes:

- VPE membership

- Instruction completion strobe.

Signals below here are part of the interface, but not used by any MIPS-designed PM:

- TC state: running, yielded, halted, suspended, waiting ITC, wait, used as SRS.

- TC running, as used by the dispatch scheduler. Note, though, that by the time the PM acts on signals like this it is always somewhat late; so it would be foolish to build hardware which attempted to respond to core signals without any "averaging".

- TC issue strobe, from DS.

- "TC has been forked". A 1-clock pulse asserted as a fork instruction completes.

- Architectural run-level (user/supervisor/kernel, exception, debug exception, error exception).

3.2.3 MIPS Policy managers included with the 1004K™ core family

MIPS Technologies will ship the core with a couple of worked-example PMs, which themselves will be useful for many purposes. You can choose between:

Equal priority (Basic round-robin)

Just wire all the TC priority group values the same, and disable the do-nothing "relax" field in *VPESchedule*. Then you'll get simple round-robin influenced only by the heuristics used by the core to keep its pipeline full.

Many applications will work just fine with this simple mechanism. You are positively recommended to avoid using anything more complicated until you really understand why!

Fixed priority

Hard-wire TC groups and priorities as required, and disable "relax". TCs of equal priority will round-robin, but the scheduler will favor higher-priority TCs.

The most likely arrangement is a two-level scheme offering higher priority for TCs to be used for threads which both (a) have real-time deadlines, and (b) can be trusted not to consume excess CPU cycles when they have no real work to do.

The "Weighted Round Robin" (WRR) policy manager

It seems like a MIPS MT CPU with two running TCs can only be told to make them equal or to give one unconditional priority over the other. You might be interested in a system which - instead - would ensure that one of the TCs consistently got more cycles than the other, but that the less-favoured TC wouldn't be starved. You can do that, by feeding the CPU with a rapidly-changing set of priorities which average out to what you want.

The building block of this is a machine which runs through a sequence of states, allowing us to provide four distinct priority "groups": other things being equal, TCs in groups 0-3 get respectively 1/15, 2/15, 4/15 and 8/15 of the CPU. In our policy manager, we can now maintain a "priority group number" for each TC and have it turned into a cycle-by-cycle priority to achieve our goal.

We run a 15-cycle counter and in each cycle of 15 award different priorities to the groups as shown in Table 3.1:

Table 3.1 Dynamic priorities for finer resolution - group priority sequences

	Priority in cycle (higher is better)														
Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Group0	0	1	0	1	0	1	0	3	0	1	0	1	0	1	0
Group1	1	0	1	3	1	0	2	1	2	0	1	3	1	0	2
Group2	2	3	2	0	2	3	1	2	1	3	2	0	2	3	1
Group3	3	2	3	2	3	2	3	0	3	2	3	2	3	2	3

The WRR PM uses dynamic priorities as shown above. You program it through the *TCSchedule* register, shown at Figure 3.2

.

Figure 3.2 Fields in the TCSchedule and VPESchedule registers (WRR policy manager)

	31	5	4	3	2	1	0
TCSchedule	0			STP	0	GRP	
VPESchedule		GPO					

TCSchedule and *VPESchedule* have lots of space for use by future (more sophisticated) policy managers. The fields defined are as follows:

STP: Stop Priority. Set 1 to prevent the associated TC running any instructions at all.

GRP: Group. Determines which of the four priority groups this TC will be in, as described above

GPO: Group Priority Override. Set 1 to disable the dynamic priority scheme. This makes this policy manager behave like the simpler fixed priority policy manager. This bit is common to both VPEs and can be set or cleared from either *VPESchedule* register.

This policy manager does not define a *VPEScheFBack* register.

TCScheFBack register

TCScheFBack counts up when any instruction is completed by this TC; it is an unsigned 32-bit value, which saturates at the maximum representable value. It is software's job to write it to zero or some other low value periodically.

WRR2 Policy Manager

The cleverly named wrr2 policy manager takes the weighted round robin policy manager from above and adds a throttling function that can deprioritize opportunistic TCs based on buffer utilization. As noted in Section 3.1.1 "Resource limits and consequences", the FSB, LDQ, and WBB are shared buffers and if they are fully occupied, critical tasks could be stalled by lower priority tasks.

This policy manager extends the capabilities of the weighted round robin policy manager (WRR). All existing functions of that policy manager are unaffected.

Internal to the CPU, there are three buffer structures which are shared by all TCs: The Load Queue (LDQ), the Fill-Store Buffer (FSB), and the Writeback Buffer (WBB). When one of these buffers becomes full, it will stall the pipeline and all TCs. In order to prevent a lower priority TC from filling one of these buffers, the policy manager can automatically lower the priority of a TC when the buffer exceeds a programmable threshold. In addition to the three internal buffers, core inputs are provided to allow an equivalent function for a system level buffer.

There are two programmable throttling functions, throttle0 and throttle1. The functions are as follows:

- Each throttle can be separately enabled for each queue/buffer.

- Software can set the threshold for enabling the throttle function.

- Software can set the group and stop priority override for each throttle.

- If both throttles are activated at the same time, throttle0 takes priority. Therefore, if both throttles are enabled, throttle0 should generally be the more restrictive one.

The function for each throttle is: When an enabled queue falls below the programmed threshold, the throttle is activated and the effective group and stop priority is overridden with the values from the throttle. When the queue goes back above the threshold, the throttle is deactivated and the group and stop priority return to the values as programmed in *TCSchedule[GRP]* and *TCSchedule[STP],* respectively.

NOTE: The throttle function is enabled when software sets any of the queue enable bits. The throttle is activated by hardware dynamically without software intervention.

This function can also be used to boost priority for threads. This is especially useful for the *PM_sys_avail* input - a TC which, when running, helps to decrease the usage of such a resource could be boosted when the resource gets too full.

Figure 3.3 Fields in TCSchedule Register (WRR2 Policy Manager)

31 24	23	22 21	20 18	17 14	13	12 11	10 8	7 4	3	2	1 0
0	T1_STP	T1_GRP	T1_TH	T1_QE	T0_STP	T0_GRP	T0_TH	T0_QE	STP	0	GRP

As noted, this policy manager builds on top of the WRR PM. The following bits behave the same as in that one:

STP: set 1 to prevent the associated TC running any instructions at all; for *VPESchedule[STP]* it disables the “relax” issue-nothing condition, which can be scheduled to save power.

GRP: determines which of the four priority groups this TC will be in, as described above; for *VPESchedule[GRP]* this is the scheduling group for the “relax” condition.

The new functionality is in the throttles. There are two throttles specified, with identical control

T0/1_QE: Throttle0/1 Queue Enable. When a bit is set in this vector, it sensitizes throttle to the available resources

Throttle 0 bit	Throttle 1 bit	Resource Monitored
7	17	PM_sys_avail: System Specific resource available
6	16	PM_fsb_avail: Fill Store Buffer entries available
5	15	PM_ldq_avail: Load Data Queue entries available
4	14	PM_wbb_avail: WriteBack Buffer entries available

T0/1_TH: Throttle0/1 Threshold. When an enabled queue input is equal to or less than this value, throttle is activated for this TC. NOTE: Setting this value to 7 will disable the threshold check and activate this throttle permanently if there are any enabled queues.

T0/T1_GRP: Throttle0/1 Group. When throttle is activated, the effective group for this TC is set to this value. (Again, if both throttles are activated, T0 takes precedence)

T0/T1_STP: Throttle0/1 Stop Priority.When throttle is active, the effective stop priority for this TC is set to this value.

.

Figure 3.4 Fields in the VPESchedule register (WRR2 policy manager)

31	5	4	3	2	1	0
	GPO					

GPO: Group Priority Override. As in the WRR policy manager, set to 1 to disable the dynamic priority scheme. The throttling function is unaffected by this bit. This bit is common to both VPEs and can be set or cleared from either *VPESchedule* register.

Figure 3.5 Fields in the VPEScheFBack register (WRR2 policy manager)

31 15	15 12	11 8	7 4	3 0
0	SYSS	WBBS	FSBS	LDQS

Since many of these buffers are configuration options that are not generally exposed to software, the *VPEScheFBack* register is used to indicate the size of the different buffers.

LDQS: Number of Load Data Queue entries

FSBS: Number of Fill Store Buffer entries

WBBS: Number of WriteBack Buffer entries

SYSS: Number of system buffer entries. This is an input to the core and the purpose and meaning of this will be system specific

TCScheFBack register

TCScheFBack counts up when any instruction is completed by this TC; it is an unsigned 32-bit value, which saturates at the maximum representable value. It is software’s job to write it to zero or some other low value periodically. This is identical to the behavior in the WRR policy manager.

3.3 Inter-thread communication storage (ITC)

ITC locations are magic memory locations used to provide low-level thread synchronization - which might be inter-thread (hence “ITC”, from “Inter-Thread Communication storage”) but could also be between customer-specific hardware and the software thread. Because ITC locations are places where threads wait for potentially long periods of time, they’re accessed - always uncached - as *Gating Storage* - described in Section 2.7.1, "Gating storage" above. The ITC block is a piece of logic outside of the 1004K core and connects through the gating storage interface. The sample ITC block connects to all cores within the cluster, enabling communication between threads running on different cores. Because it’s outside the core, SoC integrators are free to use the MIPS-supplied example logic in whole, in part, or to write their own. This section only describes the features of the sample ITC block supplied in the core package.

Each *ITC Cell* presents 32 bits of data. You should read/write these locations only as 32-bit data: partial-word loads and stores may misbehave. Be careful about compiled code too, to make sure optimization doesn’t remove or alter any load or store operations. There are 16 different “views” of the same cell, all mapped to double-word boundaries for compatibility with 64-bit implementations, so each cell occupies 16×64-bits (128 bytes) of memory space. The different views have behaviors designed to support efficient implementations of popular synchronization operations, as listed in Table 3.2. You can build your system with some or all of the ITC cells being FIFOs; to find out which cells are FIFOs look at the fields in the control view, described in Figure 3.6.

Table 3.2 ITC cell views and what they do

View offset (bytes)	**Behavior**
0	"bypass": load/store just read and write the data, without affecting the flags. If the cell is a FIFO, you write the newest entry and read the oldest (but *without* pushing the FIFO).
8	“control” view: read or write cell state, as shown in Figure 3.6.
16	"empty/full" synchronized view: the cell remembers whether anything has been written to it making it non-empty (and if it’s not a FIFO, making it full at once). Loads from an empty cell block, as do stores to a full cell. A load from a full cell makes it non-full, and (eventually, if it’s a FIFO) might empty it.
24	Empty/Full "try" (non-blocking) view. A load from an empty cell returns, but the data is always zero. A store to a full cell is quietly discarded, and the thread continues to run; but (more usefully) you can use an **`sc`** (store-conditional) instruction targeting this view and it will return 1 if the data was written, 0 if it was discarded.
32	"P/V" synchronized view: this implements a "P/V" counting semaphore. This synchronization trick was invented by Dijkstra - "p" and "v" are the "wait if zero, then count down" and "count up" functions respectively. A load from a zero cell blocks until a non-zero value appears. Otherwise the load returns the value and (atomically) decrements the stored value. Any store causes an atomic increment of the cell value, up to a maximum value of 2^{16}-1, at which it saturates. Stores never block. P/V operations do not modify the empty and full bits, which should both be cleared before an entry is used for P/V purposes. The P/V view of a FIFO location doesn’t make sense, and the result of any such access is undefined.
40	P/V "try" (non-blocking) view. Same as the synchronized P/V view, except that a load does not block, even if the cell value is zero. Again, don’t use this view for a FIFO cell
48-56	Reserved for future versions of the MIPS MT ASE.
64-120	Implementation-dependent views.

Figure 3.6 Field layout in an ITC cell control view

31 28	27 21	20 18	17	16	15 2	1	0
FIFODepth	0	FIFO_PTR	FIFO	T	0	Full	Empty

The fields in the ITC cell control view mean:

FIFODepth: indicates the number of entries in a FIFO. Zero for a non-FIFO, otherwise the FIFO has $2^{\text{FIFODepth}}$ entries.

FIFO_PTR: the index of the oldest FIFO entry (on a read that’s the next to be returned), but will read zero unless this is a FIFO cell. Write the ITC control view with *FIFO_PTR* zero, *Full* = 0 and *Empty* = 1 to reset the FIFO.

FIFO: (read-only) 1 if this cell is a FIFO (that is, it has more than one word of storage.) In the ITC implementation distributed with the 1004K family, all ITC FIFOs have four words of storage.

T: "trap" bit - causes any data access (i.e. any "empty/full" or "P/V" access) to this cell to result in an exception. Set by an OS which wants to keep track of reads and writes, perhaps because it’s recycled a TC which was waiting here and wants to know when it might have been unblocked.

Full/Empty: described in Table 3.2. There are separate full and empty bits to allow ITC cells to quietly grow into FIFOs with multiple words of storage.   Write *Empty* to 1 to reset the FIFO to a clean empty state.

3.3.1 Configuring ITC base address and cell repeat interval

The configuration information for the ITC space is held in two “tags” accessed by overloading the **`cache Index_Load_Tag_D`** and **`cache Index_Store_Tag_D`** instructions (it’s much like the mechanism used for scratchpad RAM). Set the *ErrCtl[ITC]* bit to tell the instructions to access ITC space configuration “tags”, and use addresses 0 or 8 in the **`cache`** instruction address field:

The ITC-configuration "tags" show up as in Figure 3.7.

Figure 3.7 ITC configuration information

	31 10	9 1	0
`Addr=0`	BaseAddress	0	ITC_En

	31	30 20	19 17	16 10	9 3	2 0
`Addr=8`	M	NumEntries	0	AddrMask	0	EntryGrain

What can you do with these?

BaseAddress, AddrMask: allow you to set the ITC starting physical address and region size, with at best a 1Kbyte resolution. Once this is set up and enabled, all accesses to this physical address range will go to ITC, and will no longer show up on the main system interface - so these locations will “overlay” anything else you expected to be there. Take care not to overlap any vital address.

The ITC cells can be put at any address whose alignment matches the total size of the ITC region (if you had 64 ITC cells at 256byte intervals you could place them at any 16Kbyte aligned address).

To do that set *AddrMask*:

AddrMask		*ITC region size*	*AddrMask*		*ITC region size*
`0`	=	1Kbyte	`0xF`	=	16Kbytes
`1`	=	2Kbytes	`0x1F`	=	32Kbytes
`3`	=	4Kbytes	`0x3F`	=	64Kbytes
`7`	=	8Kbytes	0x7F	=	128Kbytes

ITC_en: set 1 to use ITC - it’s zero from reset, making ITC invisible until you want it.

NumEntries: a read-only field which tells you how many 32-bit ITC cells are provided[12].

EntryGrain: let’s you control the cell spacing. Tightly spaced cells save on memory space, but widely spaced cells spread across a number of TLB pages, permitting different cells to be mapped to different processes. If you set the cell spacing very high, you’ll limit the number of cells you can access in the usual ITC region.

[12] Earlier versions of this specification used a "logarithmic" code for number of entries.

When the *EntryGrain* field is zero, cells are packed at 128-byte intervals. Other values result in cells at intervals of $128\times2^{\text{EntryGrain}}$ bytes, or:

EntryGrain		*ITC cell interval*	*EntryGrain*		*ITC cell interval*
0	=	128bytes	4	=	2Kbytes
1	=	256bytes	5	=	4Kbytes
2	=	512bytes	6	=	8Kbytes
3	=	1Kbyte	7	=	16Kbytes

To program these locations first set the *ErrCtl[ITC]* bit, which tells the cache instruction to access ITC information. Read the registers to find out how many ITC cells are available; then program your choice of cell interval and base address, with the region size set to match.

Don’t forget to clear *ErrCtl[ITC]* afterwards, so that cache operations can continue as usual.

3.4 The 1004K™ core and interrupts

As you may recall from Section 2.6.1, "Multithreading and interrupts", the interrupt system is replicated per-VPE; so the 1004K core may have two interrupt systems. Interrupt inputs (including *Int0-5*, *NMI* and the EJTAG debug interrupt *DINT*) are presented separately for the two VPEs at the core interface.

Only the internally-generated timer, fast debug channel, and performance counter overflow interrupts are always local to the VPE (you can find out what interrupt number they use by looking in the *IntCtl* register shown as Figure 7.1).

In the 1004K core any TC which is not interrupt-exempt may handle an interrupt. However, where there is a choice:

- An interrupt will be delivered to any thread which is asleep after a **`wait`** instruction (if there is one); otherwise:
- The interrupt will be delivered to any non-exempt, active thread which is not blocked waiting for a gating storage access; and only then:
- The interrupt will be delivered to an active-but-blocked thread.

See Section 7.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)" for information about the interrupt signalling and handling options that the 1004K core shares with other MIPS32 CPUs.

3.5 Synchronization: "ll" and "sc" instructions implementation

In coherent multi-processor or software multi-threaded systems, the **`ll`** and **`sc`** instructions work together to provide an RMW operation on a memory variable (with an arbitrary modification of the value) which succeeds only if it is guaranteed to have been atomic - that is, no other thread can have seen the value of the same variable between the read and the write. Moreover, **`sc`** returns a value which reports when atomicity could not be guaranteed, and the store wasn’t done; that allows software to build a retry loop to implement atomic operations. The risk of non-atomicity is detected by the cache snoop logic for cache-coherent multiprocessors, and by the intervention of an exception on software-scheduled uniprocessors.

The MIPS MT ASE requires that **`ll`**/**`sc`** also work between concurrent threads on an MT CPU. Each TC is equipped with a CP0 register called *LLAddr*, which remembers the physical address (at least to the enclosing 32-byte block) of the target location of any **`ll`**/**`sc`** sequence. The 1004K core detects possible non-atomicity by checking every write made by any thread against the *LLAddr* of all other TCs.

The hardware keeps a single bit of state per TC called a "link bit" - the link bit is not directly visible to software. The link bit is most often zero, but is set by a **`ll`** instruction and then cleared by any condition threatening atomicity. It’s cleared if:

- An external intervention of the types associated with a store hits on the address. Reads from other cores will not clear the link bit

- Some other TC’s store is to the same block as our *LLAddr*;

- An **`eret`** instruction runs for this TC’s VPE (which means there’s been an exception, which could mean this TC has been rescheduled in the middle of its sequence);

- Some other software wrote this TC’s *TCRestart* register to cause it to execute elsewhere. This is to catch conditions where OS software running on some other thread "reschedules" the TC: we don’t want the link bit to survive such indignities.

Then the **`sc`** succeeds only if the link bit is still set when it executes.

In the MIPS MT ASE the **`sc`** instruction is also used (in this case independently of **`ll`** or the link bit) to provide feedback from a store to an ITC location which might fail: see Section 3.3, "Inter-thread communication storage (ITC)".

Chapter 4

Initialization 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.

4.1 Probing your CPU - Config CP0 registers

The four registers *Config* and *Config1-3* are 32-bit CP0 registers which contain information about the CPU's capabilities. *Config1-3* are strictly read-only. The few writable fields in *Config* — notably *Config[K0]* — are there for historic compatibility, and are typically written once soon after bootstrap and never changed again.

The 1004K CPU also defines *Config7* for some implementation-specific settings (which most programmers will never use).

Broadly speaking the registers have these roles:

Table 4.1 Roles of Config registers

Config	A mix of historical and CPU-dependent information, described in Figure 4.1 below. Some fields are writable.
Config1	Read-only, strictly to the MIPS32 architecture. *Config1* shows the primary cache configuration and basic CPU capabilities, while *Config2* shows information about L2 and L3 caches, if fitted (the L2 cache is optional and the L3 cache is unavailable in 1004K family CPUs). Shown in Figure 4.2 and Figure 4.3 below.
Config2	
Config3	Read-only, strictly to Release 2 of the [MIPS32] architecture. More CPU capability information.
Config6	Provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2* and *Config3*.
Config7	1004K-CPU-specific, with both read-only and writable fields. It's a strong convention that the writable fields should default to "expected" behavior, so beginners may simply leave these fields alone. The fields are described later, in Section C.4.5 "The Config7 register".

While initializing your CPU, you might also want to look at the *EBase* register, which can be used to relocate your exception entry points: see Figure 7.2 and the text round it.

4.1.1 The Config register

Figure 4.1 Fields in the Config Register

31	30 28	27 25	24	23	22	21	20	19	18	17	16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	ISP	DSP	UDI	SB	0	WC	MM	0	BM	BE	AT	AR	MT	0	VI	K0
1	2	2						0	1				0	1			0	2

In Figure 4.1:

M: reads 1 if *Config1* is available (it always is).

K23, KU, K0: set the cacheability attributes of chunks of the memory map by writing these fields. All share a 3-bit encoding with the cacheability field found in TLB entries, which is described in Table 5.3 in Section 5.4.2 “Cacheability options”.

Config[K0] sets the cacheability of kseg0, but it would be very unusual to make that anything other than cacheable (on different, cache-coherent CPUs, it may want to be set to cacheable-coherent). The power-on value of this standard field is not mandated by the [MIPS32] architecture; but the 1004K CPU follows the recommendation to set it to "2", making "kseg0" *uncached*. That can be surprising; early system initialization software typically re-writes it to "3" in order that *kseg0* will be cached, as expected.

If your 1004K CPU-based system uses fixed mapping instead of having a TLB, *Config[K23]* is for program addresses 0xC000.0000-0xFFFF.FFFF (the “kseg2” and “kseg3” areas), while *Config[KU]* is for program addresses 0x0000.0000-0x7FFF.FFFF (the “kuseg” area). If you have a TLB, these regions are mapped and these fields are unused (write only zeroes to them).

ISP, DSP: read 1 if I-side and/or D-side scratchpad (SPRAM) is fitted, see Section 5.6, "Scratchpad memory/ SPRAM".

(Don’t confuse this with the MIPS DSP ASE, whose presence is indicated by *Config3[DDSP]*.)

UDI: reads 1 if your CPU implements user-defined "CorExtend" instructions, and if the CorExtend unit is made available to this VPE by the setting of the *VPEConf0* register. “CorExtend” is available on CPUs whose name ends in "Pro".

SB: read-only "SimpleBE" bus mode indicator. If set, means that this CPU will only do simple partial-word transfers on its OCP interface; that is, the only partial-word transfers will be byte, aligned half-word and aligned word.

If zero, it may generate partial-word transfers with an arbitrary set of bytes enabled (which some memory controllers may not like).

WC: **Warning**: this is a diagnostic/test field, not intended for customer use, and may vanish without notice from a future version of the CPU.

Set this 1 to make the *Config1[IS]* and *Config1[DS]* fields writable, which allows you to reduce the number of available L1 I- and D-cache ``sets per way", and shrink the usable cache size. You'd never want to do this in a real system, but it is conceivable it might be useful for debug or performance analysis. If you have an L2 cache configured, then this makes *Config2[SS]* writable in the same way.

MM: writable: set 1 if you want writes resulting from separate store instructions in write-through mode merged into a single (possibly burst) transaction at the interface. This has no affect on cache writebacks (which are always whole blocks together) or uncached writes (which are never merged).

Note that the *Config[MM]* bit is not replicated per-VPE (like most CP0 fields): there’s only one per CPU and anything written by one VPE affects the other one.

BM: read-only - tells you whether your bus uses sequential or sub-block burst order; set by hardware to match your system controller.

BE: reads 1 for big-endian, 0 for little-endian.

AT: MIPS32 or MIPS64 compliance On 1004K family CPUs it will read “0”, but the possible values are:

0	MIPS32
1	MIPS64 instruction set but MIPS32 address map
2	MIPS64 instruction set with full address map

AR: Architecture revision level. On 1004K family CPUs it will read “1”, denoting release 2 of the MIPS32 specification.

0	MIPS32/MIPS64 Release 1
1	MIPS32/MIPS64 Release 2

MT: MMU type (all MIPS Technologies CPUs may be configured as type 1 or 3):

0	None
1	MIPS32/64 compliant TLB
2	“BAT” type
3	MIPS-standard fixed mapping

VI: 1 if the L1 I-cache is virtual (both indexed and tagged using virtual address). No contemporary MIPS Technologies CPU has a virtual I-cache.

K0: as described in the notes above on Config[K23] etc, this field determines the cacheing behaviour of the fixed kseg0 memory region .

4.1.2 The Config1-2 registers

These two read-only registers tell you the size of the TLB, and the size and organization of L1, L2 and L3 caches (a zero “line size” is used to indicate a cache which isn’t there). They’re best described together.

Config1 has some fields which tell you about the presence of some of the older extensions to the base MIPS32 architecture are implemented on this CPU. These bits ran out, and other extensions are noted in *Config3*.

Figure 4.2 Fields in the Config1 Register

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMUSize	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP
1			4	3		4	3		0	1	1	1	1	

Figure 4.3 Fields in the Config2 Register

31	30 28	27 24	23 20	19 16	15 13	12	11 8	7 4	3 0
M	TU	TS	TL	TA	SU	L2B	SS	SL	SA
1	0	0	0	0	0	0			

Config1[M]: continuation bit, 1 if *Config2* is implemented.

Config1[MMUSize]: the size of the TLB array (the array has MMUSize+1 entries). On this multithreading CPU this is a read-only field which automagically returns the number of entries available to your VPE - unless the TLB is shared, in which case it returns the size of the whole array.

Config1[IS,IL,IA,DS,DL,DA]: for each cache this reports

S	Number of sets per way. Calculate as: 64×2^S
L	Line size. Zero means no cache at all, otherwise calculate as: 2×2^L
A	Associativity/number of ways - calculate as A + 1

So if (IS, IL, IA) is (2,4,3) you have 256 sets/way, 32 bytes per line and 4-way set associative: that’s a 32Kbyte cache.

Config1[C2,FP]: 1 if coprocessor 2 or or an FPU (coprocessor 1) fitted, respectively. A coprocessor 2 would be a customer-designed coprocessor. In a multithreading CPU these bits reflect whether the units are really available to this VPE, which depends on the setting of *VPEConf0[NCP2,NCP1]*.

Config1[MD]: 1 if MDMX ASE is implemented in the floating point unit (very unlikely for the 1004K CPU).

Config1[PC]: there is at least one performance counter implemented, see Section 10.4 “Performance counters”.

Config1[WR]: reads 1 if the 1004K CPU has watchpoint registers, see Section 10.3 “CP0 Watchpoints”.

Config1[CA]: reads 1 if the MIPS16e compressed-code instruction set is available (as it generally is on MIPS Technologies CPUs).

Config1[EP]: reads 1 because an EJTAG debug unit is always provided, see Section 10.1, "EJTAG on-chip debug unit".

Config2[M]: continuation bit, 1 if *Config3* is implemented.

Config2[TU]: implementation-specific bits related to tertiary cache, if fitted. Can be writable.

Config2[TS,TL,TA]: tertiary cache size and shape - encoded just like *Config1[IS,IL,IA]* which see above.

Config2[SU]: implementation-specific bits for secondary cache, if fitted. Can be writable.

Config2[L2B]: Set to disable L2 cache ("bypass mode"). Setting this bit also forces *Config2[SL]* to 0 — most OS code will conclude that there isn't an L2 cache on the system, which can be useful.

Writing this bit controls a signal out to the L2 cache hardware. However, reading it does not read back what you just wrote: it reflects the value of a signal sent back from the L2 cache. With MIPS Technologies' L2 cache logic, that feedback signal will reflect the value you just wrote, with some implementation-dependent delay (it's unlikely to be 100 cycles, but it could easily be more than 10). For more details refer to [L2CACHE].

Config2[SS,SL,SA]: secondary cache size and shape, encoded like *Config1[IS,IL,IA]* above.

4.1.3 The Config3 register

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture. A few of them were in *Config2*, but that ran out of bits.

Figure 4.4 Config3 Register Format

31	30	29	28 14	13	12	11	10	9	8 7	6	5	4	3	2	1	0
M	0	CMGCR		ULRI	0	DSP2P	DSPP	CTXTC	0	VEIC	VInt	SP	CDMM	MT	SM	TL
0							1				1	0		1	0	

Fields shown in Figure 4.4 include:

Config3[M]: continuation bit which is zero, because there is no *Config4*.

Config3[CMCGR]: reads 1 if Global Control Register in the Coherence Manager are implemented and the *CMGCRBase* register is present. Reads 0 otherwise*Config3[ULRI]*: reads 1 if the CPU implements the *UserLocal* register, typically used by software threads packages. More information in Section C.4.2 "The UserLocal register".

DSP2P, DSPP: *DSPP* reads 1 if the MIPS DSP extension is implemented — as described in Chapter 9, "The MIPS32® DSP ASE" on page 127. *DSP2P* reads 0 — it distinguishes CPUs which conform to the later revision 2 of the DSP ASE.

CTXTC: reads 1 when the *ContextConfig* register is implemented. The width of the *BadVPN2* field in the *Context* register depends on the contents of this register.

VEIC: read-only bit from the CPU input signal *SI_EICPresent* which should be set in the SoC to alert software to the availability of an EIC-compatible interrupt controller, see Section 7.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)".

VInt: reads 1 when the 1004K CPU can handle vectored interrupts.

SP: reads 0 when the 1004K CPU does not support sub-4Kbyte page sizes.

CDMM: reads 0 when the 1004K CPU does not support the Common Device Memory Map.

MT: reads 1 to show that 1004K family CPUs implement the MIPS MT (multithreading) extension.

SM: reads 0, the 1004K CPU does not handle instructions from the "SmartMIPS" ASE.

TL: reads 1 if your CPU is configured to do instruction trace.

4.1.4 The Config6 register

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2* and *Config3*.

Figure 4.5 Config6 Register Format

31 15	14	13	12 10	9	8	7 2	1	0
0	SPCD	SYND	IFUPerfCtl	NMRUP	NMRUD	0	JRCP	JRCD

SPCD disables performance counter clock shutdown. The primary use of this bit is to keep performance counters alive when the CPU is in sleep mode.

SYND disables Synonym tag update. By default, all synonym load misses will opportunistically update the tag so that subsequent loads will hit at lookup.

IFUPerfCtl encodes IFU events that provide debug and performance information for the IFU pipeline.

NMRUP indicates that a Not Most Recently Used JTLB replacement scheme is present.

NMRUD disables the Most Recently Used JTLB replacement scheme bit.

JRCP indicates that a JR Cache is implemented.

JRCD indicates that JR Cache Prediction is enabled.

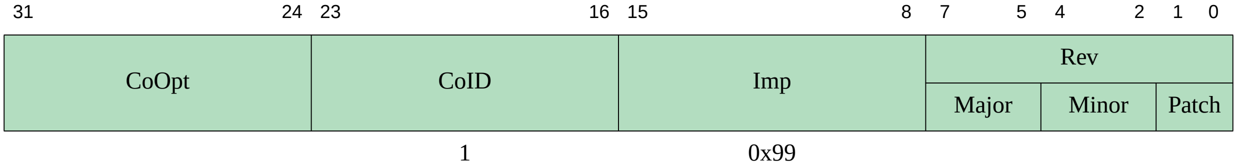
4.1.5 CPU-specific configuration — Config7

Config7 is packed with implementation-specific fields. Most of the time, you leave them alone (a few of them might sometimes need to be set as required by your SoC designer). So we’ve left these registers defined in the all-CP0 appendix, in Section C.4.5 “The Config7 register”.

4.2 PRId register — identifying your CPU type

This register identifies the CPU to software. It’s appropriately printed as part of the start-up display by any software telling the world about the CPU on start-up; but when portable software is configuring itself around different CPU attributes, it's always preferable to sense those attributes directly — look in other *Config* registers, or perhaps use a directed software probe.

Figure 4.6 Fields in the PRId Register



PRId[CoOpt]: Whatever is specified by the SoC builder who synthesizes the CPU — refer to your SoC manual. It should be a number between 0 and 127 — higher values are reserved by MIPS Technologies.

PRId[CoID]: Company ID, which in this case is "1" for MIPS Technologies Inc.:

PRId[Imp]: Identifies the particular processor, which in this case is 0x99 for the 1004K family. Any processor with different CP0 features must have a new *PRId* field.

PRId[Rev]: The revision number of the CPU design, used to index entries in errata lists etc. By MIPS Technologies' convention the revision field is divided into three subfields: a major and minor number; with a nonzero "patch" revision number is for a release with no functional change. CPU licensees can consult [ERRATA] for authoritative information about the revision IDs associated with releases of the 1004K CPU.

The following incomplete and possibly not up-to-date table of historical revisions is provided as a guide to programmers who don't have [ERRATA] on hand:

Table 4.2 1004K™® CPU releases and PRId[Revision] fields

Release Identifier	*PRId[Revision]* Maj.min.patch/hex	Description	Date
2_0_*	1.0.0 / 0x20	First (GA) release of the 34K CPU	September 30, 2005
2_1_*	2.1.0 / 0x44	MR1 release. Bug fixes, 8KB cache support.	March 10, 2006
2_2_0	2.2.0 / 0x48	Allow up to 9 TCs, alias-free 64KB L1 D-cache option.	August 31, 2006
2_2_1	2.2.1 / 0x49	Enable use of MIPS SOC-it® L2 Cache Controller.	October 12, 2006
2_3_*	2.3.0 / 0x4c	Less interlocks round **`cache`** instructions, relocatable reset exception vector location.	January 3, 2007
2_4_*	2.4.0 / 0x50	New *UserLocal* register, alias-proof I-cache hit-invalidate operation, can **`wait`** with interrupts disabled, per-TC performance counters.	October 31, 2007
2_5_*	2.5.0/0x54	Errata fixes	January, 2009
1_0_*	1.0.0/0x20	First (GA) release of the 1004K CPU	June 30, 2008
1_1_*	1.1.0/0x24	Errata fixes	January, 2009
1_2_*	1.2.0/0x28	Feature updates: improved low power support, fast debug channel, on-chip PDtrace buffers	July, 2009
2_0_*	2.0.0 / 0x40	General availability of 24K CPU.	March 19, 2004
3_0_*	3.0.0 / 0x60	COP2 option improvements.	September 30, 2004

Table 4.2 1004K™® CPU releases and PRId[Revision] fields

3_2_*	3.2.0 / 0x68	PDtrace available.	March 18, 2005
3_4_*	3.4.0 / 0x6c	ISPRAM (I-side scratchpad) option added	June 30, 2005
3_5_*	3.5.0 / 0x74	8KB cache option	December 30, 2005
3_6_*	3.6.0 / 0x78	L2 support., 64KB alias-free D-cache option, option to have up to 8 outstanding cache misses (previous maximum 4).	July 12, 2006
3_7_*	3.7.0 / 0x7c	Less interlocks round **cache** instructions, relocatable reset exception vector location.	January 3, 2007
4_0_*	4.0.0 / 0x80	New *UserLocal* register, alias-proof I-cache hit-invalidate operation, can **wait** with interrupts disabled.	October 31, 2007
4_1_*	4.1.0/0x84	Errata fixes	January, 2009
2_0_*	2.0.0 / 0x40	General availability of 24KE CPU.	June 30, 2005
2_1_*	2.1.0 / 0x44	8KB cache option	December 30, 2005
2_2_*	2.2.0 / 0x48	L2 support., 64KB alias-free D-cache option, option to have up to 8 outstanding cache misses (previous maximum 4).	July 12, 2006
2_3_*	2.3.0 / 0x4c	Less interlocks round **cache** instructions, relocatable reset exception vector location.	January 3, 2007
2_4_*	2.4.0 / 0x50	New *UserLocal* register, alias-proof I-cache hit-invalidate operation, can **wait** with interrupts disabled.	October 31, 2007
2_5_0	2.5.0/0x54	Errata fixes	January, 2009
1_0_*	1.0.0 / 0x20	Early-access release of 74K family RTL.	January 31, 2007
2_0_0*	2.0.0 / 0x40	First generally-available release of 74K family CPU.	May 11, 2007
2_1_0*	2.1.0 / 0x44	Can **wait** with interrupts disabled.	October 31, 2007

4.3 Multi-Threaded bootstrap issues

You are likely to deal with MIPS MT features at three stages as you boot the system:

- Boot the system, probably without dependence on the MIPS MT extension. It's good if first-level system bootstrap (which is bound to be awkward, system-dependent code) is not also sensitive to changes in the CPU feature set.

  So the first-level bootstrap typically wants to make sure that any new multithreading behavior is suppressed until wanted: see Section 4.3.1, "Bootstrapping without worrying about multithreading".

- Set up VPEs and TCs.

  Once you reach the point where you're running software which wants to exploit multithreading, you need to discover what resources the CPU has, and to set them up. That's described in Section 4.3.2, "Configuring your choice of VPEs and TCs".

  Special care should be taken when you're initializing a VPE which is to run non-MT-aware "legacy" software - perhaps a whole legacy operating system: notes in Section 4.3.3, "Setting up a VPE for legacy software"

  If you are running co-operative software on two VPEs and are able to make minor changes to the source code, it will usually be more efficient to share the TLB entries (the "legacy-ready" approach is really a hard-wired partition of the entries): see Section 4.3.4, "Sharing and not sharing the TLB".

- Thread initialization for explicit multi-threading, see Section 4.3.5, "Setting up a TC to run a thread".

4.3.1 Bootstrapping without worrying about multithreading

It's usually going to make sense to deal with the complexities of multi-threading only at the point in the system where you start to use the facilities. For many systems that means that initial bootstrap software (perhaps a boot monitor or reset-time diagnostic) will be better off ignoring multithreading.

Fortunately that's straightforward. A MIPS MT CPU comes out of reset with just TC #0 running, affiliated to VPE #0, and looking single-threaded. Moreover, the *VPEConf0[MVP]* bit is set, so the bootstrap software is all-powerful and can do whatever is required to set up the right VPEs and threads for the system.

To help out with post-mortem diagnosis of software errors, most per-TC registers belonging to TCs other than TC#0 are left unchanged by reset. In particular, *TCRestart* is unaffected, so you get some idea of where various threads had got to.

Now bootstrap your computer. If the software needs to know it, it can read its own TC and VPE number from *TCBind*.

As always, bootstrap software is responsible for initializing CP0 registers; a register may only be skipped if you are certain that random contents in it will not disrupt your software.

4.3.2 Configuring your choice of VPEs and TCs

When you arrive at that software which wants to start an extra VPE or TC, you first need to discover what resources your CPU has. The pre-multithreading *Config* and *Config1-3* registers used to tell you everything; but in a MT CPU, those registers reflect just one VPE's resources, which in turn depend on what has been configured through *MVPControl* and *VPEConf0-1* (details in Figure 2.5 and Figure 2.6 and the notes to them) In fact, since in the 1004K core initialization software has a fair amount of control over what resources are allocated to each VPE, some fields in the previously read-only *Config* registers are *writable*. However, that's only done where necessary: for example, since all VPEs share the caches, all VPEs can and do use read-only cache information from *Config1-2*.)

The total CPU resources are enumerated in *MVPConf0-1*, which you can see in detail in Figure 2.4 .

Getting a second VPE/TC into use

Suppose you have some software loaded into memory, but you want to start running it with TC #1 bound to VPE #1. Currently both the per-VPE and per-TC registers and other resources are in their post-reset state: the critical ones need software initialization before they can start.

Only a thread with *VPEConf0[MVP]* set can do this - fortunately VPE #0 will come out of reset with *MVP* set (if you already cleared it, you've resigned. Reset the CPU and try again!). Then set *MVPControl[VPC]*.

Set *VPEControl[TargTC]* to 1, the other TC's number, so you can write the other TCs registers with **`mttr`**.

You certainly don't want the other VPE running while you do this sort of thing, and you should clear your *VPEControl[EVP]* bit while you're working. You should probably use the **`dvpe`**/**`mtc0`** instructions as a "bracket", rather than manipulating the *EVP* bit directly. So your overall flow should be like this:

```
dvp t0
ehb                  # execution hazard barrier, make sure dvp takes effect
...
set MVPControl[VPC]
(initialise VPE #1 and TC #1)
...
mtc0 t0, VPEControl  # undo the dvp
```

OK, so now let's look at how you "initialize VPE #1 and TC #1".

From now on **`mttr`** will operate on the TC of your choice. You'll probably want to do quite a lot of set-up of both per-TC and per-VPE fields

So for TC#1:

- Set *TCHalt*. In fact TC#1 can't run anything yet, because you're still under **`dvpe`** control - but the MT specification allows CPUs to treat the "halted" state specially. Don't omit this.

- TC #1's VPE affiliation may not be set as you wish, so set *TCBind[CurVPE]* to 1 (the other VPE's number)

- A word of warning. This section lists all the "important" fields. When any MIPS CPU is powered up, only a rather small set of CP0 register fields are initialized. But when a MIPS MT CPU is powered up, only fields for VPE #0 and TC #0 are initialized at all. Your new VPE may have random garbage in any writable CP0 field. So if yours may be the first use of that VPE from power-on, iterate over all the CP0 registers setting all writable fields to "safe" values.

- Set *TCStatus*. *TCStatus[A]*:will have to be set to 1 so the TC can run (this bit is the "allocated" bit for **`fork`**, and is required when setting up a thread manually.) All other bits can be zero, at least to start with - once you get something working, though, go back to the detailed description in for any other fields in Section 2.9.4, "TCStatus" on page 35.

- Set *TCRestart* to the program location where you want your new thread to start.

Now let's look at VPE#1's registers:.

- Set *VPEControl* zero, and set *VPEConf0* to leave *VPEConf0[XTC]* = 1 (it should match your affiliated TC number, and must do so if you want to start the software “single-threaded”), *VPEConf0[MVP]* = 0, *VPEConf0[VPA]* = 1. For full details consult Figure 2.1 and Figure 2.6 and their notes.

- Set VPE #1/TC #1’s *Status* register: If your intention is to have VPE #1’s software mimic "coming out of reset" you might want to set its *Status[EXL]* set 1, so it starts in exception mode. Think about *Status[BEV]* - if set and your new thread takes an exception (which quite often happens with brand new code, due to one slip or another), then with *BEV* set it will use the ROM exception vectors, which are always shared with VPE #0 - and might not be what you wanted. On the other hand, if you do clear *Status[BEV]*, make sure you’ve set up code to catch any exceptions delivered at the non-ROM address.You might also set *EBase* to give VPE #1 different exception entry points from VPE #0 (unless you really want to share them).

- Set VPE #1’s *VPEConf1* register. *VPEConf1[NCP2, NCP1, NCX]* determine whether your new VPE will be able to use coprocessors 2 and 1 (CP1 is the floating point unit) and the UDI instruction set, respectively. If the co-processor has only one bank of registers, you may well want to deny use of the co-processor to all but one of the VPEs.

At the end of our sequence you re-enable multithreading (by restoring the old value of *VPEControl*). Your last step is to use **`mttr`** to write TC #1’s *TCHalt* to zero. Now TC#1 should start up and start running your code.

4.3.3 Setting up a VPE for legacy software

In general you can support only *one* piece of legacy software on a 1004K family core. The VPEs see the same basic MIPS architecture memory map, and a few things are commonly shared - not least the exception entry points.

Your legacy software has to be told (by some means, beyond the scope of this manual) not to use all the physical memory in the system. Most likely the new MT-aware software will also need to use some virtual memory in the kseg0/kseg1 regions, too.

The "legacy" VPE needs to be carefully set up to fool the old software into seeing and using a congenial MIPS32 CPU. That means:

1. Set up this VPE with just one TC;

2. You’ll initialize all the relevant new MIPS MT registers and resources to keep the legacy software happy for its lifetime. Consult the full list of registers in Section 2.9, "Multithreading ASE - CP0 (privileged) registers".

4.3.4 Sharing and not sharing the TLB

It’s not really visible to software whether there is really more than one TLB in any 1004K core, but you can software configure it so that you get either:

- *Hard partition*: each VPE appears to have its own TLB fully compatible with the MIPS32 architecture (the sizes of the VPE’s TLBs are as set when the SoC was built - so while this will often be half-size each, it may not be); OR

- *Shared*: the VPEs share all the TLB’s entries (up to a maximum of 64 entries). This is certainly a good choice if one of the VPEs doesn’t use the TLB at all (which is not unusual in many legacy embedded systems.)

  But it is also particularly likely to be a good choice if the VPEs are to run the *same* software and that software is under your control; for example, if you’re using them to run a close approximation to a dual-CPU SMP Linux OS (a *VSMP* system.)

But to share the TLB you will need to make some modifications to the TLB maintenance code, as described below.

To partition the TLB just set *MVPControl[STLB]* zero. Each VPE's *Config1[MMUSize]* field will show the size allocated to it, as configured by your SoC designer.

To share the TLB set *MVPControl[STLB]* to 1. It will probably be convenient to set *Config1[MMUSize]* to show the full array. A change to *STLB* should be made only by "unmapped" code, and with the TLB empty of valid entries.

You don't usually need to make any change to the critical TLB refill exception handler, so long as - as is usual - it relies on random replacement (that is, the update to the TLB is done with a **tlbwr** instruction.) The TLB CP0 registers used in a typical TLB miss handler include *Context*, *EntryHi*, *EntryLo0-1* and *PageMask*. All are replicated for each VPE.

The *Random* register is handled specially. **tlbwr** will not use a value for *Random* which coincides with the other VPE's value of the manually-set TLB index register *Index*.

Meanwhile, other kernel software may be doing software-driven updates to the TLB (mostly that means removing entries). The TLB maintenance software will need to run single-threaded, at least in part. There are three possible sources of unwanted concurrency, and software has to attend to two of them:

1. The other VPE may itself be performing some TLB maintenance. This can be fixed with a one-thread-at-a-time software semaphore, just like the ones you use in an SMP OS.

2. Another TC belonging to the same VPE may get a TLB-related exception. This can be fixed by bracketing critical parts of the TLB maintenance routine with a **dmt**/**mtc0** pair, disabling all TC-level parallelism while the operation is completed.

3. A TC belonging to the other VPE may get a TLB-related exception. But the hardware makes this OK. The only resources the two VPEs share are the TLB entries itself, and the only entry the other VPE will access is the one used by **tlbwr** as indexed by the *Random* register. The hardware will ensure that the *Random* value used by the other VPE will be different from the *Index* value you're using for your maintenance routine. So no software fix is needed.

For efficient use of TLB entries, maintenance software should return *Index* to an unused value (which represents no entry) as soon as it has finished - otherwise you're blocking *Random* from selecting some particular TLB slot. The recommended value is 0x8000.0000; the top bit of *Index* is writable on MIPS MT CPUs for this purpose.

There's another more subtle change. The TLB is used early in the pipeline to translate instruction addresses. A TLB-related exception ("TLB Invalid" for example) detected at this stage is not taken until and unless the instructions are scheduled into the main pipeline. By that time many instructions from other threads may have gone past, and perhaps one of them may have done a refill which dislodged the invalid TLB entry. So the TLB invalid exception handler might find there's no translation entry in the TLB at all: your best bet, in that case, is probably to just return from the exception without doing anything, which will lead to a TLB miss exception and all should get fixed up.

4.3.5 Setting up a TC to run a thread

The easiest way to set a previously-unoccupied TC running a thread is to use the **fork** instruction.

To prepare to use **fork** you need to make sure there is at least one TC with the *TCStatus[DA]* bit set to 1 (which indicates it's available for fork), the bit *TCStatus[A]* zero (i.e. the TC is not already in use), and *TCHalt* zero.

However, there's nothing to prevent you from setting up a TC manually; just set the thread restart address *TCRestart* and set *TCStatus[A]*. You should almost always set *TCHalt* before doing manual adjustments on a TC, and clear it when you've finished. If the TC started to run (perhaps an interrupt routine) while being worked on it would be likely to lead to confusion.

4.3.6 TCs recycled as Shadow registers

The MIPS32 architecture permits CPUs to be configured such that a particular interrupt handler (or in some cases all exception handlers) should be invoked with a complete alternate set of general-purpose registers: a *Shadow register set*. That allows you to write a very low-overhead handler, because you don't have to save the interrupted thread's registers.

There are some applications where explicit multithreading will fix your problem better than shadow registers. But there are other cases where you really want shadow registers rather than multiple TCs, and the 1004K core gives you the choice - you can close down a TC for thread business, and make its registers available for shadow set use. See Section 7.4.1, "Recycling multi-threading CPU's TCs as shadow sets".

Chapter 5

Memory map, caching, reads, writes and translation

In this chapter:

- Section 5.1, "The memory map": basic memory map of the system.
- Section 5.3, "Reads, writes and synchronization"
- Section 5.4, "Caches"
- Section 5.6, "Scratchpad memory/SPRAM": optional on-chip, high-speed memory (particularly useful when dual-ported to the OCP interface).
- Section 5.8, "The TLB and translation": how translation is done and supporting CP0 registers.

5.1 The memory map

A 1004K CPU system can be configured with either a TLB (virtual memory translation unit) or a fixed memory mapping, or even with one VPE using the TLB and one with fixed mapping.

A TLB-equipped VPE sees the memory map described by the [MIPS32] architecture (which will be familiar to anyone who has used a 32-bit MIPS architecture CPU) and is summarized in Table 5.1. The TLB gives you access to a full 32-bits physical address on the system interface. More information about the TLB in Section 5.8, "The TLB and translation".

Table 5.1 Basic MIPS32® architecture memory map

Segment Name	*Virtual range*	*What happens to accesses here?*
kuseg	`0x0000.0000-0x7FFF.FFFF`	The only region accessible to user-privilege programs. Mapped by TLB entries.
kseg0	`0x8000.0000-0x9FFF.FFFF`	a fixed-mapping window onto physical addresses 0x0000.0000-0x1FFF.FFFF. Almost invariably cacheable - but in fact other choices are available, and are selected by *Config[K0]*, see Figure 4.1. Accessible only to kernel-privilege programs.
kseg1	`0xA000.0000-0xBFFF.FFFF`	a fixed-mapping window onto the same physical address range 0x0000.0000-0x1FFF.FFFF as “kseg0” - but accesses here are uncached. Accessible only to kernel-privilege programs.
kseg2 sseg	`0xC000.0000-0xDFFF.FFFF`	Mapped through TLB, accessible with supervisor or kernel privilege (hence the alternate name).
kseg3	`0xE000.0000-0xFFFF.FFFF`	Mapped through TLB, accessible only with kernel privileges.

5.2 Fixed mapping option

To save chip area for applications not needing a full TLB, threads in one or both VPEs can use a simple fixed mapping ("FMT") memory translator, which plays the same role. You can find out whether a VPE has fixed mappings by reading the CP0 field *Config[MT]* (see Figure 4.1 and descriptions). With the fixed mapping option, virtual address ranges are hard-wired to particular physical address windows, and cacheability options are set through CP0 register fields as summarized in Table 5.2:

Table 5.2 Fixed memory mapping

Segment Name	*Virtual range*	*Physical range*	*Cacheability bits from*
kuseg	`0x0000.0000-0x7FFF.FFFF`	`0x4000.0000-0xBFFF.FFFF`	*Config[KU]*
kseg0	`0x8000.0000-0x9FFF.FFFF`	`0x0000.0000-0x1FFF.FFFF`	*Config[K0]*
kseg1	`0xA000.0000-0xBFFF.FFFF`	`0x0000.0000-0x1FFF.FFFF`	(uncached)
kseg2/3	`0xC000.0000-0xFFFF.FFFF`	`0xC000.0000-0xFFFF.FFFF`	*Config[K23]*

Even in fixed-mapping mode, the cache parity error status bit *Status[ERL]* still has the effect (required by the MIPS32 architecture) of taking over the normal mapping of "kuseg"; addresses in that range are used unmapped as physical addresses, and all accesses are uncached, until *Status[ERL]* is cleared again.

5.3 Reads, writes and synchronization

The MIPS architecture permits implementations a fair amount of freedom as to the order in which loads and stores appear at the CPU interface. Most of the time anything goes: so long as the software behaves correctly, the MIPS architecture places few constraints on the order of reads and writes seen by some other agent in a system.

5.3.1 Read/write ordering and cache/memory data queues in the 1004K™ CPU

To understand the timing of loads and stores (and sometimes instruction fetches), we need to say a little more about the internal construction of the 1004K CPU. In order to maximize performance:

- *Loads are non-blocking*: execution continues "through" a load instruction, and only stops when the program tries to use the GPR value it just loaded.
- *Writes are "posted"*: a write from the CPU is put aside (the hardware stores both address and data) until the CPU can get access to the system interface and send it off.
- *Cache refills are completed "opportunistically"*: the CPU may still be running on from a non-blocking load or prefetch when data arrives back from the cache. The data required to make good a miss is forwarded to the relevant GP register, so the returning data is not urgently needed in the cache. The data waits until a convenient moment before it gets put into the cache line.

All of these are implemented with "queues", called the LDQ, WBB and FSB (for "fill/store buffer" — it's used both for writes which hit and for refills after a cache miss) respectively. All the queues handle data first-come, first served. The WBB and FSB queues need to be *snooped* - a subsequent store to a location with a load pending had better not be allowed to go ahead until the original load data has reached the cache, for example. So each queue entry is tagged with the address of the data it contains.

An LDQ entry is required for every load that misses in the cache. Moreover, an LDQ entry must be available for any load - even if it will hit in the cache, the logic requires that the LDQ entry is available if needed. This queue allows the

CPU to keep running even though there are outstanding loads. When the load data is finally returned from the system, the LDQ and the main CPU logic act together to write this data into the correct GPR (which will then restart the program, if it was blocked on an attempt to use this register).

The WBB (Write Back Buffer) queue holds data waiting to be sent out over the system interface, either from D-cache writebacks or uncached/write-through store instructions.

FSB (Fill Store buffer) queue entries are used to hold data that is waiting to be written into the D-cache. An FSB entry gets used during a cache miss (when it holds the refill data), or a write which hits in the cache (when it holds the data the CPU wrote). Loads and stores snoop the FSB so that accesses to lines “in flight” can be dealt with correctly.

All this has a number of consequences which may be visible to software:

- *Number of non-blocking loads which may be pending*: the CPU has either four or nine LDQ entries according to configuration (but it’s always at least one per TC). That limits the number of outstanding loads. As mentioned above, you can’t start a load - even one which will in fact hit in the cache - unless you have a free LDQ entry.

- *Hit-under-miss*: the D-cache continues to supply data on a hit, even when there are outstanding misses with data in flight. FSB entries remember the in-flight data. So it is quite normal for a read which hits in the cache to be “completed” - in the sense that the data reaches a register - before a previous read which missed.

- *Write-under-miss*: the CPU pipeline continues and can generate external store cycles even though a read is pending, so long as WBB slots are available. The 1004K CPU’s “OCP” interface is non-blocking too (reads consist of separate address and data phases, and writes are permitted between them), so this behavior can often be visible to the system.

- *Miss under miss*: the 1004K CPU can continue to run until the pending read operations exhaust FSB or LDQ entries. More often, of course, it will try to use the data from the pending miss and stall before it gets that far.

- *CPU interface ordering*: at the CPU interface, read operations may be split into an address phase and a later data phase, with other bus operations in between.

  The 1004K CPU - as is permitted by [MIPS32] - makes only limited promises about the order in which reads and writes happen at the system interface. In particular, uncached or write-through writes may be overtaken by cache line reads triggered by a load/store cache miss *later* in sequence. However, uncached reads and writes are always presented in their program sequence (program sequence inside a thread). When some particular program needs to do things “really in order”, the **`sync`** instruction can help, as described in the next section.

Cache management operations interact with several queues: see Section 5.4.6 “L1 Cache instruction timing”.

5.3.2 The “sync” instruction in 1004K™ family CPUs

If you want to be sure that some other agent in the system sees a pair of transactions to uncached memory in the order of the instructions that caused them, you should put a **`sync`** instruction between the instructions. Other MIPS32/64-compliant CPUs may reorder loads and stores even more; portable code should use **`sync`**[13].

But sometimes it’s useful to know more precisely what **`sync`** does on a particular CPU. On 1004K **`sync`**:

[13] Note that **`sync`** is described as only working on “uncached pages or cacheable pages marked as coherent”. But **`sync`** also acts as a synchronization barrier to the effects produced by routine cache-manipulation instructions - hit-writeback and hit-invalidate.

- Stalls until all loads, stores, refills are completed and all write buffers are empty (that is until the LDQ, FSB and WBB are empty);

- In some systems the CPU will also generate a synchronizing transaction on the OCP system interface if *Config7[ES]* bit is set[14]. Not all systems do this. See Section C.4.5 "The Config7 register" for more details.

5.3.3 Write gathering and "write buffer flushing" in 1004K™ family CPUs

We mentioned above that writes to the system (whether uncached writes or cache write-backs) are performed somewhat lazily, the write being held in the WBB queue until a convenient moment. That can have two system-visible effects:

- Writes can happen later than you think. Your write will happen before the next uncached read or write, but that's all you know. To make sure that a write has gone out on the OCP bus you can use a **`sync`** (as above): but that meaning of **`sync`** is CPU-dependent, so that code is non-portable. And your write might still be posted somewhere in a system controller, unless you know your system is built to prevent it. Sometimes it's better to code a dummy uncached read from a nearby location (which will "flush out" buffered writes on pretty much any system).

- If your cache is configured for write-through, then cached writes to locations in the same "cache line"-sized chunk of memory may be gathered - stored together in the WBB, and then dealt with by a single "wider" OCP write than the one you originally coded. Sometimes, this is what you want. When it isn't, put a **`sync`** between your successive writes. Regular uncached writes are never merged, but special "uncached accelerated" writes may be — see Section 5.4.3 below.

5.4 Caches

Most of the time caches just work and are invisible to software... though your programs would go twenty times slower without them. But this section is about when caches aren't invisible any more.

Like most modern MIPS CPUs, the 1004K CPU has separate primary I- and D-caches. They are virtually-indexed and physically-tagged, so you may need to deal with *cache aliases*, see Section 5.4.10, "Cache aliases". The design provides for 8Kbyte, 16Kbyte, 32Kbyte or 64Kbyte caches; but the largest of those are likely to come with some speed penalty. The 1004K CPU's primary caches are 4-way set associative.

Your 1004K CPU can optionally be built with a L2 (level 2 or secondary) cache. see section below for details.

But don't hard-wire any of this information into your software. Instead, probe the *Config1* register defined by [MIPS32] (and described in the notes to Figure 4.2) to determine the shape and size of the L1 and any L2 cache.

5.4.1 The L2 cache option

The L2 cache is an option available to your SoC builder. Basic facts and figures:

- The L2 cache is attached to the CPU's standard 64-bit OCP system interface, and when you fit it everything else is attached to the CPU *through* the L2 cache, which has a system-side interface for that purpose. The CPU-side

[14] This will be a read with the signal *OC_MReqInfo[3]* set. Handling of this transaction is system dependent, but a typical system controller will flush any external write buffers and complete all pending transactions before telling the CPU that the transaction is completed. Ask your system integrator how it works in your SoC.

interface is enhanced and augmented to support **`cache`** instructions targeted at the L2, and to carry back performance counter information and so on.

- The L2 cache is located between the Coherence Manager and main memory. A 256-bit interface exists between the blocks to make the most of the available L2 bandwidth.

- The L2 ‘s size can be 128Kbytes, 256Kbytes, 512Kbytes or 1Mbyte. However, there are options which allow the SoC builder to have one or more of the ways of the cache memory array visible as normal system memory instead. There’s very little in this manual about that option. — see [L2CACHE].

- The L2 cache is indexed and tagged with the physical address, so is unaffected by cache aliases.

- Cache lines are either 32 bytes long (matching the L1 caches) or 64 bytes. The L2 cache’s memories are accessed 256 bits at a time internally, though it has 64-bit interfaces.

- It can be configured with 4-way or 8-way set-associative organization. In a 4-way cache the line replacement policy is “least recently used” (LRU); true LRU is impractical for an 8-way set associative cache, so something simpler (a “pseudo-LRU”) is used.

- The cache has an option for error detection and correction. 1-bit data errors can be corrected and all 2-bit errors detected with an 8-bit-per-doubleword ECC field. Check bits are provided on cache tags, too. If your L2 has ECC fitted, *ErrCtl[L2P]* will be writable — see Section 5.4.18 “ErrCtl register” for details.

- The cache is write-back but does not allocate a line on a write miss (write miss data is just sent directly to the system memory). It is write-through for memory regions which request that policy -- see Section 5.4.2 “Cacheability options” for details.

- The L2 cache can run synchronously to the CPU CPU, but (particularly for memory arrays larger than 256Kbytes) would typically then be the critical path for timing. It will more often use a 1:2 or 2:3 clock ratio. The L2’s far-side OCP interface may run at any of a wide range of ratios from the L2 clock down.

- In an effort to keep everything going the cache manages multiple outstanding transactions (it can handle as many as 15 outstanding misses). Misses are resolved and responses sent as they happen, not in the order of presentation.

- Latency: the L2 logic allows the memory access to be pipelined, a reasonable choice for larger or slower arrays: ask your SoC builder. The L2 delivers hit data in a burst of four 64-bit doublewords. The first doubleword appears after 9 or 10 L2 clocks (10 for pipelined-array systems) and the rest of the burst follows on consecutive clocks. Added to this is some extra time taken for the original L1 miss to be discovered, synchronizing to the L2 clock, and returning the data to the CPU: typically, add 5 CPU clocks.

  An L2 miss is slightly more expensive than an L1 miss from the same memory, since we don’t start the memory access until we’ve discovered that the data isn’t in the L2. The L2 memory interface can be configured to be 64-bit or 256-bit wide. An L2 miss will deliver miss data to the CPU CPU in burst of four 64-bit doublewords .Because the CPU connects to the rest of the system through the L2 cache, it also adds 4 L2 cycles to the latency of all transactions which bypass the L2.

- The L2 cache requires software management, and you can apply the same **`cache`** instructions to it as to the L1 D-cache.

5.4.2 Cacheability options

Any read or write made by the 1004K CPU will be cacheable or not according to the virtual memory map. For addresses translated by the TLB the cacheability is determined by the TLB entry; the key field appears as *EntryLo[C]*. Table 5.3 shows the code values used in *EntryLo[C]* - the same codes are used in the *Config* entries used to set the behavior of regions with fixed mappings (the latter are described in Table 5.2.)

Some of the undefined cacheability code values are reserved for use in cache-coherent systems.

Table 5.3 Cache Code Values

Code	Cached?	How it Writes	Notes
0	cached	write-through	An unusual choice for a high-speed CPU, probably only for debug
2	uncached		
3	cached	writeback	All normal cacheable areas
7	uncached	"Uncached Accelerated"	Unusual and interesting mode for high-bandwidth write-only hardware; see Section 5.4.3, "Uncached accelerated writes". Such writes just bypass the L2 cache, if there is one.

5.4.3 Uncached accelerated writes

The 1004K CPU 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. Sequential word stores in such regions are gathered into cache-line-sized chunks, before being written with a single burst cycle on the CPU interface.

Such regions are uncached for read, and partial-word or out-of-sequence writes have "unpredictable" effects - don't do them. The burst write is normally performed when software writes to the last location in the memory block or does an uncached-accelerated write to some other block; but it can also be triggered by a **`sync`** instruction, a **`pref nudge`**, a matching load or any exception. If the block is not completely written by the time it's pushed out, it will be written using a series of doubleword or smaller write cycles over the 1004K CPU's 64-bit memory interface.

If you have an L2 cache, regions marked as "uncached accelerated" are L2-uncached.

5.4.4 The cache instruction and software cache management

The 1004K CPU's caches are not fully "coherent" and require OS intervention at times. The **`cache`** instruction is the building block of such OS interventions, and is required for correct handling of DMA data and for cache initialization. Historically, the **`cache`** instruction also had a role when writing instructions (unless the programmer takes some action, those instructions may only be in the D-cache whereas you need them to be fetched through the I-cache when the time comes). But where possible use **`synci`** for that purpose, as described in Section 5.4.8 "Cache management when writing instructions - the "synci" instruction".

A cache operation instruction is written **`cache op,addr`** where **`addr`** is just an address format, written as for a load/store instruction. Cache operations are privileged and can only run in kernel mode (**`synci`** works in user mode, though). Generally we're not showing you instruction encodings in this book (you have software tools for that stuff) but in this case it's probably necessary, so take a look at Figure 5.1.

31 26	25 21	20 18	17 16	15 0
cache	base	op		offset
47	register	what to do	which cache	

Figure 5.1 Fields in the encoding of a cache instruction

The `op` field packs together a 2-bit field which selects which cache to work on:

0	L1 I-cache
1	L1 D-cache
2	reserved for L3 cache
3	L2 cache

and then adds a 3-bit field which encodes a command to be carried out on the line the instruction selects.

Before we list out the individual commands in Table 5.4; the cache commands come in three flavors which differ in how they pick the cache entry (the “cache line”) they will work on:

- *Hit-type cache operation*: presents an address (just like a load/store), which is looked up in the cache. If this location is in the cache (it “hits”) the cache operation is carried out on the enclosing line. If this location is not in the cache, nothing happens.

- *Address-type cache operation*: presents an address of some memory data, which is processed just like a cached access - if the cache was previously invalid the data is fetched from memory.

- *Index-type cache operation*: as many low bits of the address as are required are used to select the byte within the cache line, then the cache line address inside one of the four cache ways, and then the way. You have to know the size of your cache (discoverable from the *Config1-2* registers, see the notes to Figure 4.3) to know exactly where the field boundaries are, but your address is used something like this:

31		5	4 0
Unused	Way1-0	Index	byte-within-line

  Beware: the MIPS32 specification leaves CPU designers to choose whether to derive the index from the virtual or physical address. Don’t leave it to chance: with index-type operations use a kseg0 address, so that the virtual and physical address are the same (at least apart from some high bits which certainly won’t affect any cache index). This also avoids a potential pitfall related to cache aliases.

  The L1 caches are 4-way set-associative, so data from any given address has four possible cache locations - same index, different value of the “Way1-0” bits as above.

Don’t define your own C names for cache manipulation operation codes, at least not if you can use a standard header file from MIPS Technologies on open-source terms: see [m32c0 h].

5.4.5 Cache instructions and CP0 cache tag/data registers

MIPS Technologies’ CPUs use different CP0 registers for cache operations targeted at different caches. That’s already quite confusing, but to make it more interesting these registers have somehow got different names — those

Table 5.4 Operations on a cache line available with the cache instruction

Value	Command	What it does
0	Index invalidate	Sets the line to "invalid". If it's a D-cache or L2 cache line which is valid and "dirty" (has been written by CPU since fetched from memory), then write the contents back to memory first. This is the best and simplest way to invalidate an I-cache when initializing the CPU - though if your cache is parity-protected, you also need to fill it with good-parity data, see Fill below. This instruction is not suitable for initializing caches, where it might cause random write-backs: see the **`Index Store Tag`** type below.
1	Index Load Tag	Read the cache line tag bits and addressed doubleword data into the *TagLo* etc registers (see Table 5.1 for names). Operation for diagnostics and geeks only.
2	Index Store Tag	Set the cache tag from the *TagLo* registers. To initialize a writable cache from an unknown state, set the *TagLo* registers to zero and then do this to each line.
3	Index Store Data	Write cache-line data. Not commonly used for caches, but it is used for management of scratchpad RAM regions described in Section 5.6 "Scratchpad memory/SPRAM".
4	Hit invalidate	hit-type invalidate - do not writeback the data even if dirty. May cause data loss unless you know the line is not dirty. Certain CPUs implement a special form of the I-side hit invalidate, where multiple searches are done to ensure that any line matching the effective physical address is invalidated (even if it doesn't match the supplied virtual address for page color) — see Section 5.4.10 "Cache aliases" below.
5	*Sorry, different meanings for code "5" on L1 I-cache.*	
	Writeback invalidate	On the L1D-cache or L2 cache: (hit-type operation) invalidate the line but only after writing it back, if dirty. This is the recommended way of invalidating a writable line in a running cache.
	Fill	On an L1 I-cache: (address-type operation) fill a suitable cache line from the data at the supplied address - it will be selected just as if you were processing an I-cache miss at this address. Used to initialize an I-cache line's data field, which should be done when setting up the CPU when the cache is parity protected.
6	Hit writeback	If the line is dirty, write it back to memory but leave it valid in the cache. Used in a running system where you want to ensure that data is pushed into memory for access by a DMA device or other CPU.
7	Fetch and Lock	An address-type operation. Get the addressed data into the same line as would be used on a regular cached reference (if the data wasn't already cached that might involve writing back the previous occupant of the cache line). Then lock the line. Locked lines are not replaced on a cache miss. It stays locked until explicitly invalidated with a cache An attempt to lock the last entry available at some particular index fails silently.

used here and in C header files. I hope Table 5.1 helps. In the rest of this document we'll either use the full software name or (quite often) just talk of *TagLo* without qualification.:

Table 5.1 Caches and their CP0 cache tag/data registers

Cache	CP0 Registers	CP0 number
L1 I-cache	*ITagLo*	*28.0*
	IDataLo	*28.1*
	IDtataHi	*29.1*
L1 D-cache	*DTagLo*	*28.2*
	DDataLo	*28.3*
L2 cache	*L23TagLo*[1]	*28.4*
	L23DataLo	*28.5*
	L23DataHi	*29.5*

[1] In past versions of this manual *L23TagLo* was known as "STagLo", and so on. But this name is more mnemonic.

5.4.6 L1 Cache instruction timing

Most CP0 instructions are used rarely, in code which is not timing-critical. But an OS which has to manage caches around I/O operations or otherwise may have to sit in a tight loop issuing hundreds of **`cache`** operations at a time, so performance can be important. Firstly, any D-side **`cache`** instruction will check the FSB queue (as described in Section 5.3 "Reads, writes and synchronization") for potentially matching entries[15]. The "potential match" check uses the cache index, and avoids taking any action for most irrelevant FSB activity. But on a potential match the cacheop waits (stalling the whole CPU pipeline) while any pending cache refills happen, and while any dirty lines evicted from the cache are sent out at least to the CPU's write buffer. Typically, this will not take more than a few clocks.

Once this is done, hit-type **`cache`** instructions which miss in the cache and therefore do nothing (and that's probably much the commonest case) run through the pipeline with no delay. Instructions which take some action, though, stall the pipeline and delay all subsequent instructions by a few cycles. The various possibilities are shown in Table 5.5.

[15] In earlier versions of the 24K and 34K family CPUs, no index check is performed and *any* D-side cacheop waits until the FSB is empty. There are unusual conditions where this can noticeably impact performance.

Table 5.5 Cache instruction timings.

Operation	Line State	Action	Delay (CPU cycles)
Hit Invalidate	×	Invalidate cache line, no memory traffic	3
Hit writeback	Clean	Nothing happens	4
	Dirty	Write back cache line	8
Hit writeback invalidate	Clean	Invalidate line	5
	Dirty	Write back line and invalidate	8
Index Store Tag		Update tag	3
Fetch and lock	Hit	Line is in cache, just lock it	3
	Miss	Line has to be fetched into cache, and this is a blocking operation. Wait for that then add...	7

5.4.7 L2 cache instruction timing

The L2 cache run synchronously with the CPU but at a configurable clock ratio. The L2 operations will be significantly slower than L1 versions even at the same clock ratio. Exactly how slow is dependent on the performance of the memory blocks used to build your L2 cache and the L2 clock ratio.

5.4.8 Cache management when writing instructions - the "synci" instruction

The **synci** instruction (new to the MIPS32 Release 2 update) provides a clean mechanism - available to user-level code, not just at kernel privilege level - for ensuring that instructions you've just written are correctly presented for execution (it combines a D-cache writeback with an I-cache invalidate). You should use it in preference to the traditional alternative of a D-cache writeback followed by an I-cache invalidate.

synci does nothing to an L2 cache — the L2 cache is unified, and there's no need to do anything special there to make data visible for instruction fetch.

5.4.9 Cache management and multithreaded CPUs

The cache management registers are all replicated per-VPE but not per-TC, so obviously you have to avoid multiple threads on the same VPE attempting to use cache operations concurrently.

Moreover, in 1004K family CPUs the two VPEs share the cache. In general write-back operations and the kind of invalidate which automatically writes-back a dirty line may be safely run by either VPE at any time. All other operations may cause undesirable effects unless you make sure they're done by only one VPE at a time; and in particular, you should get the cache initialized by one VPE running alone.

There are also some corner cases which can lead to short-term unfair scheduling of two threads which are concurrently using I-side cacheops. The hardware is designed to ensure that such operations don't overlap, so if two threads A and B do **cache** instructions concurrently, thread B will be stopped. As is usual in the 1004K CPU's pipeline, thread B is temporarily suspended with the intention of re-issuing its last couple of instructions when the I-cache is free again. But with bad luck and the ill-timed intervention of thread C (which would have to be running largely from cache), our thread's cacheop can find itself persistently scheduled after another thread which is making heavy use of cache operations, and may be repeatedly pushed back. At worst it may make no progress until the other cacheop-using thread has moved on to other activities.

There are some other obscure cases where threads are suspended when CPU resources run out, where the same sort of thing can theoretically happen: instructions dependent on a `div` result, or a load/store waiting for an FSB slot.

5.4.10 Cache aliases

A cache alias is when one physical address can exist in two or more indices within a cache. This can happen because the cache arrays are virtually indexed and depending on the cache way size and page size, some of those index bits may be translated. With the expense of added cache invalidates, it is possible for software to work around aliases. The expense of maintaining that workaround gets unbearable when it is spread across multiple CPUs. Thus, the 1004K CPU has added hardware to eliminate the aliases, or at least the visible effects of them.

For alias-prone data cache sizes (32KB and 64KB), the cache arrays are banked on the low order physical address bits. This effectively makes the data cache physically indexed and thus alias-free. On the instruction cache, aliases are primarily a problem when trying to invalidate an entry. If you are going to the trouble of invalidating a cache line, you would generally want all copies of that line to be invalidated. When an address based invalidate is requested, the cache controller will automatically check all possible locations where it could be and invalidate them all.

The 1004K CPU has L1 caches which are virtually indexed but physically tagged. Since it’s quite routine to have multiple virtual mappings of the same physical data, it’s possible for such a cache to end up with two copies of the same data. That becomes troublesome:

- *When you want to write the data*: if a line is stored in two places, you’ll only update one of them and some data will be lost (at least, there’s a 50% chance it will be lost!) This is obviously disastrous: systems generally work hard to avoid aliases in the D-cache.

- *When you want to invalidate the line in the cache*: there’s a danger you might invalidate one copy but not the other. This (more subtle) problem can affect the I-cache too.

It can be worked around. There’s no problem for different virtual mappings which generate the same cache index; those lines will all compete for the 4 ways at that index, and then be correctly identified through the physical tag.

The 1004K CPU’s smallest page size is 4Kbytes, that’s 2^{12} bytes. The paged memory translation means that the low 12 bits of a virtual address is always reproduced in the physical address. Since a 16Kbyte, 4-way set-associative, cache gets its index from the low 12 bits of the address, the 16Kbyte cache is alias-free. In general, you can’t get aliases if each cache “way” is no larger than the page size.

In 32Kbyte and 64Kbyte caches, one or two top bits used for the index are not necessarily the same as the corresponding bits of the physical address, and aliases are possible. The value of the one or two critical virtual address bits is sometimes called the *page color.*

It’s possible for software to avoid aliases if it can ensure that where multiple virtual mappings to a physical page exist, they all have the same color. An OS can do that by enforcing virtual-memory alignment rules (to at least a 16Kbyte boundary) for shareable regions. It turns out this is practicable over a large range of OS activities: sharing code and libraries, and deliberate interprocess shared memory. It is not so easy to do in other circumstances, particularly when pages to be mapped start their life as buffers for some disk or network operation[16]...

[16] There’s a fair amount of rather ugly code in the MIPS Linux kernel to work around aliases. D-cache aliases (in particular) are dealt with at the cost of quite a large number of extra invalidate operations.

So the 1004K CPU contains logic to make a 32Kbyte or 64Kbyte D-cache alias-free (effectively one or two index bits are from the physical address, and used late in the cache access process to maintain performance). This logic is a build option, and *Config7[AR]* flag should read 1 if your CPU was built to have an alias-free D-cache.

A 32Kbyte or 64Kbyte I-cache is subject to aliases. It's not immediately obvious why this matters; you certainly can't end up losing writes, as you might in an alias-prone D-cache. But I-cache aliases can lead to unexpected events when you deliberately invalidate some cache content using the **`cache`** instruction. An invalidation directed at one virtual address translated to a particular physical line may leave an undesirable valid copy of the same physical data indexed by a virtual alias of a different color. To solve this, some 1004K CPUs are built to strengthen hit-type I-cache invalidate instructions (those include hit-type **`cache`** instructions and the **`synci`** instruction), so as to guarantee that no copy of the addressed physical line remains in the cache. This facility is available if the *Config7[IAR]* bit reads 1; but if it's available but your software doesn't need it, you can restore "legacy" behavior by setting *Config7[IVA]* to 1. The *Config7[AR]* and *Config7[IAR]* bits indicate that these alias removal features are present on the CPU.Refer to Section C.4.5 "The Config7 register" for details.

The MIPS Technology supplied L2 cache (if configured) is physically indexed and physically tagged, so does not suffer from aliases.

5.4.11 Cache locking

[MIPS32] provides for a mechanism to lock a cache line so it can't be replaced. This avoids cache misses on one particular piece of data, at the cost of reducing overall cache efficiency.

Caution: in complex software systems it is hard to be sure that cache locking provides any overall benefit - most often, it won't. You should probably only use locking after careful measurements have shown it to be effective for your application.

Lock a line using a **`cache FetchAndLock`** (it will not in fact re-fetch a line which is already in the cache). Unlock it using any kind of relevant **`cache`** "invalidate" instruction[17] - but note that **`synci`** won't do the job, and should not be used on data/instruction locations which are cache-locked.

5.4.12 Cache initialization and tag/data registers

The cache tag/data registers (listed out in Table 5.1) are used for staging tag information being read from or written to the cache (the 1004K CPU has no "TagHi" registers, which are only needed for CPUs with a bigger physical address range). [MIPS32] declares that the contents of these registers is implementation dependent, so they need some words here.

ITagLo is used for the I-cache, *DTagLo* for the D-cache, and *L23TagLo* for the L2 cache, if configured. Some other MIPS CPUs use the same staging register for all caches, and initialization software written for such CPUs is not portable to the 1004K CPU.

Before getting into the details, note that it's a strong convention that you can write all-zeros to the appropriate *TagLo* register and then use **`cache IndexStoreTag`** to initialize a cache entry to a legitimate (but empty) state. Your cache initialization software should rely on that, not on the details of the registers.

Only diagnostic and test software will need to know details; but Figure 5.2 shows all the fields:

[17] It's possible to lock and unlock lines by manipulating values in the *TagLo* register and then using a **`cacheIndex_Load_Tag`** instruction... but highly non-portable and likely to cause trouble. Probably for diagnostics only.

Figure 5.2 Fields in the TagLo Registers

31 12	11 10	8	7	6	5	4 1	0
TagLo	U	0	V	D	L	0	P

ITagLo and *DTagLo* can be used in a special mode; when *ErrCtl[WST]* is 1, the appropriate *TagLo* register's fields change completely, as shown in Figure 5.8 and its notes below. But let's look at the standard fields first:

TagLo: the cache address tag — the low 12 bits of the address are implied by the position of the data in the cache.

×: a field not described for the 1004K CPU but which might not always read zero.

V: 1 when this cache line is valid.

D: 1 when this cache line is dirty (that is, it has been written by the CPU since being read from memory).

L: 1 when this cache line is locked, see Section 5.4.11, "Cache locking".

P: parity bit for tag fields other than the *TagLo[D]* bit, which is actually held separately in the "way-select" RAM. When you use the *TagLo* register to write a cache tag with **`cache IndexStoreTag`** the *TagLo[P]*: bit is generally not used - instead the hardware puts together your other fields and ensures it writes correct parity. However, it is possible to force parity to exactly this value by first setting *ErrCtl[PO]*.

E: always 0

P0: parity bit for tag fields other than the *TagLo[D]* bit, which is actually held separately in the "way-select" RAM. When you use the *TagLo* register to write a cache tag with **`cache IndexStoreTag`** the *TagLo[P]*: bit is generally not used - instead the hardware puts together your other fields and ensures it writes correct parity. However, it is possible to force parity to exactly this value by first setting *ErrCtl[PO]*.

5.4.13 L23TagLo Regiser

This register in the 1004K CPU is implemented to support access to external L2 cache tags via **`cache`** instructions. The definition of the fields of this 32 bit register are defined by the SoC designer. Refer to the section on L2 Transactions in the document ““MIPS32® 1004K$^{\text{CoreTrade}}$ Processor CPU Family Integrator's Guide, MD00499” for further information on using this register.

Figure 5.3 L23TagLo Register Format

31 0
DATA

5.4.14 L23DataLo Register

On 1004K family CPUs, test software can read or write cache data using a **`cache`** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 5.4 L23DataLo Register Format

31	0
DATA	

Table 5.6 L23DataLo Register Field Description

Fields				
Name	Bit(s)	Description	Read / Write	Reset State
DATA	31:0	Low-order data read from the cache data array.	R/W	Undefined

5.4.15 L23DataHi Register

On 1004K family CPUs, test software can read or write cache data using a **`cache`** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 5.5 L23DataHi Register Format

31	0
DATA	

Table 5.7 L23DataHi Register Field Description

Fields				
Name	Bit(s)	Description	Read / Write	Reset State
DATA	31:0	High-order data read from the cache data array.	R/W	Undefined

5.4.16 TagLo registers in special modes

The usual *TagLo* register fields are a view of the underlying cache tags. But load-tag/store tag cacheops act differently in special modes activated by setting bits in *ErrCtl* (see Section 5.4.18 “ErrCtl register” for details):

- When *ErrCtl[SPR]* is set, the L1 *TagLo* registers are used to configure scratchpad memory, if fitted. That’s described in Section 5.6 “Scratchpad memory/SPRAM” below, where you’ll find a field diagram for the TagLo registers in that mode.

- When *ErrCtl[WST]* is set, the tag registers are used to provide diagnostic/test software with direct read-write access to the “way select RAM” — parts of the cache array. This is highly CPU-dependent and is described in Section C.4.7 “Cache registers in special diagnostic modes”.

5.4.17 Parity error exception handling and the CacheErr register

The 1004K CPU does not check parity on data (or control fields) from the external interface - so this section really is just about parity protection in the cache. It’s a build-time option, selected by your system integrator, whether to include check bits in the cache and logic to monitor them.

At a system level, a cache parity exception is usually fatal - though recovery might be possible sometimes, when it is useful to know that the exception is taken in “error mode” (that is, *Status[ERL]* is set), the restart address is in *ErrorEPC* and you can return from the exception with an **`eret`** — it uses *ErrorEPC* when *Status[ERL]* is set.

But mainly, diagnostic-code authors will probably find the *CacheErr* register's extra information useful.

Figure 5.6 Fields in the CacheErr Register

31	30	29	28	27 26	25	24	23	22	21 20	19	18 0
ER	EC	ED	ET	0	EB	EF	SP	EW	Way	0	Index
					For L2 cache errors only				Way		

ER: was the error on an I-fetch (0) or on data (1)? Applicable only to L1 cache errors.

EC: in L1 cache (0) or L2-or-higher cache (1)?

ED,ET: 1 for error in data field/tag field respectively.

Not Supported

Not Supported

EB: 1 if data and instruction-fetch error reported on same instruction, which is unrecoverable. If so, the rest of the register reports on the instruction-fetch error.

On an L2 error: 1 if an error occurred in more than one of the cache's memory arrays if *EC* is also set— the hardware manual [SUM] renames this field as *CacheErr[EM]*. The rest of the register can only reflect information about one of the errors: it shows tag errors as highest priority, then data, then way-select.

EF: unrecoverable (fatal) error (other than the *EB* type above). Some parity errors can be fixed by invalidating the cache line and relying on good data from memory. But if this bit is set, all is lost... It's one of the following:

1. Line being displaced from cache ("victim") has a tag parity error, so we don't know whether to write it back, or whether the writeback location (which needs a correct tag) would be correct.

2. The victim's tag indicates it has been written by the CPU since it was obtained from memory (the line is "dirty" and needs a write-back), but it has a data parity error.

3. Writeback store miss and *CacheErr[EW]* error.

4. At least one more cache parity error happened concurrently with or after this one, but before we reached the relative safety of the cache parity error exception handler.

If the *EC* bit is set this bit is referring to the errors in L2 (external) cache.

SP: error affecting a scratchpad RAM access, see *Section 5.6, "Scratchpad memory/SPRAM"* below.

EW: parity error on the "dirty" (cache modified) or way-selection bits. This means loss of LRU information, which — most of the time — is recoverable.

Way: the way-number of the cache entry where the error occurred. **Caution**: for the L1 caches (which are no more than 4-way set associative) this is a two-bit field. But an L2 cache might be more highly set-associative, and then this field grows *down*. In particular, MIPS' (possibly 8-way set associative) L2 cache uses a 3-bit *Way* field as shown.

Index: the index (within the cache way) of the cache entry where the error occurred... except that the low bits are not meaningful. The index is aligned as if it's a byte address, which is good because that's what Index-type **`cache`** instructions need. It resolves the failing doubleword for a data error, or just the failing line for a tag error. We've shown a 14-bit field, because that's large enough to provide the index for the 1004K CPU's largest configurable (4 ways by 16KB) L1 cache option.

Two other fields are related to the processing of cache errors. Other implementations have laid claim to all of the bits in this register, so these bits were relegated to the *ErrCtl* register. The FE and SE bits in that register are used to detect nested cache errors and are described in the next section.

If you want to study this error further, you'll probably use an index-type **`cache`** instruction to read out the tags and/or data. The cache instruction's "index" needs the way-number bits added to *CacheErr[Index]*'s contents; see Figure 5.1 and its notes above for how to do that.

5.4.18 ErrCtl register

This register has two distinct roles. It contains "mode bits" which provide different views of the *TagLo* registers when they're used for access to internal memory arrays and cache diagnostics. But it also controls parity protection of the caches (if it was configured in your CPU in the first place).

Figure 5.7 Fields in the ErrCtl Register

31	30	29	28	27	26	25	24	23	22 21	20	19	18 13	12	11 4	3 0
PE	PO	WST	SPR		ITC	LBE	WABE	L2P	0	SE	FE		0	PI	PD
0	0	0	0	0	0				0	0	0				

Two fields are 'overflow' from the CacheErr register and relate to the error state:

FE/SE: Used to detect nested errors. FE (FirstError) is set on any cache error. SE (Second Error) is set when an error is seen and FE is set. Software should clear FE once it has logged enough error information that taking a second error will not be fatal.

The rest of the fields can be summarized as such: running software should set just the parity enable (*PE*) bit to enable cache parity checking as required, and leave it zero otherwise. The fields are as follows:

PE: 1 to enable cache parity checking. Hard-wired to zero if parity isn't implemented.

PO: (parity overwrite) - set 1 to set the parity bit regardless of parity computation, which is only for diagnostic/test purposes.

After setting this bit you can use **`cache IndexStoreTag`** to set the cache data parity to the value currently in *ErrCtl[PI]* (for I-cache) or *ErrCtl[PD]* (for D-cache), while the tag parity is forcefully set from *TagLo[P]*.

WST: test mode for **`cache IndexLoadTag`**/**`cache IndexStoreTag`** instructions, which then read/write the cache's internal "way-selection RAM" instead of the cache tags.

SPR: when set, index-type **`cache`** instructions work on the scratchpad/SPRAM, if fitted - see Section 5.6, "Scratchpad memory/SPRAM".

PI/PD: parity bits being read/written to caches (I- and D-cache respectively).

ITC: set to make **`cache IndexLoadTag`**/**`cache IndexStoreTag`** operate on the control/configuration "tags" for ITC storage locations - see Section 3.3.1 "Configuring ITC base address and cell repeat interval".

LBE, WABE: field indicating whether a bus error (the last one, if there's been more than one) was triggered by a load or a write-allocate respectively: see below. Where both a load and write-allocate are waiting on the same cache-line refill, both could be set. These bits are "sticky", remaining set until explicitly written zero.

L2P: Controls ECC checking of an L2 cache, if it's fitted and has that capability.

For backward-compatibility, you only set *L2P* when you want to make a different error-checking choice at the L1 and L2 levels. So L2 error checking is enabled if *ErrCtl[PE,L2P]* == 01 or *ErrCtl[PE,L2P]* == 10.

5.5 Bus error exception

The CPU's "OCP" hardware interface rules permit a slave device attached to the system interface to signal back when something has gone wrong with a read. This should not be used to report a read parity error; if parity is checked externally, it would have to be reported through an interrupt. Typically a bus error means that some subsystem has failed to respond. Bus errors are not signalled on an OCP write cycle, and (if they were) the 1004K CPU ignores them.

Instruction bus error exceptions are precise (when the exception happens *EPC* always points to the instruction where fetch failed). But a data-side bus error is usually caused by a load, and the (non-blocking) load which caused it may have happened a long time before the busy cycle finishes and the error is signalled. So a bus error exception caused by a load or store is *imprecise*; *EPC* does not necessarily (or even usually) point to the instruction causing the memory read. In fact, the exception may not be taken by the TC which ran the load/store instruction: all we know is that the exception will be delivered to some TC affiliated to the same VPE. It's even possible for a bus error on a cache refill to affect two threads (any number of threads could be waiting on the arrival of the same cache line). In this obscure case an exception will be delivered to some TC in each affected VPE.

On a load the hardware knows which TC or TCs were waiting for the load which went wrong, and the *TCBind[TBE]* bit will be set for each suffering TC.

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.

There is an obscure corner case. The 1004K CPU's D-cache is "write-allocate": so a write which misses in the cache will trigger a read, to fill the cache line ready to receive the new data. If you're unlucky enough to get a bus error on that read-for-refill, the bus error will be associated with a store. After a bus error you can look at *ErrCtl[LBE]*/ *ErrCtl[WABE]* to see whether the error was caused by a load or write-allocate.

5.6 Scratchpad memory/SPRAM

The 1004K CPU (like most of MIPS Technologies' CPUs) can be equipped with modestly-sized high speed on-chip data memory, called *scratchpad RAM* or *SPRAM*. SPRAM is connected to a cache interface, alongside the I- and/or D-cache, so is available separately for the I- and D-side (*ISPRAM* and *DSPRAM*).

MIPS Technologies provide the interface on which users can build many types and sizes of SPRAM. We also provide a "reference design" for both ISPRAM and DSPRAM, which is what is described here. If you keep the programming interface the same as the reference design, you're more likely to be able to find software support. The reference design allows for on-chip memories of up to 1Mbytes in size.

There are two possible motives for incorporating SPRAM:

SPRAM can be made larger than the maximum cache size.

Even for smaller sizes, it is possible to envisage applications where some particularly heavily-used piece of data is well-served by being permanently installed in SPRAM. Possible, but unusual. In most cases heavily-used data will be handled well by the D-cache, and until you really know otherwise it's better for the SoC designer to maximize cache (compatible with his/her frequency needs.)

But there's another more compelling use for a modest-size SPRAM:

- *"DMA" accessible to external masters on the OCP interface*: the SPRAM can be configured to be accessible from an OCP interface. OCP masters will see it just as a chunk of memory which can be read or written.

  Because SPRAM stands in for the cache, data passed through the SPRAM in this way doesn't require any software cache management. This makes it spectacularly efficient as a staging area for communicating with complex I/O devices: a great way to implement "push" style I/O (that is where the device writes incoming data close to the CPU).

SPRAM must be located somewhere within the physical address map of the CPU, and is usually accessed through some "cached" region of memory (uncached region accesses to scratchpad work with the 1004K reference design, but may not do so on other implementations - better to access it through cacheable regions). It's usually better to put it in the first 512Mbytes of physical space, because then it will be accessible through the simple kseg0 "cached, unmapped" region - with no need to set up specific TLB entries.

Because the SPRAM is close to the cache, it inherits some bits of cache housekeeping. In particular the **`cache`** instruction and the cache tag CP0 registers are used to provide a way for software to probe for and establish the size of SPRAM[18].

Probing for SPRAM configuration

The presence of scratchpad RAM in your CPU is indicated by a "1" bit in one or both of the CP0 *Config[ISP,DSP]* register flags described in Figure 4.1. The MIPS Technologies reference design requires that you can query the size of and adjust the location of scratchpad RAM through "cache tags".

To access the SPRAM "tags" (where the configuration information is to be found) first set the *ErrCtl[SPR]* bit (see Section 5.4.18 "ErrCtl register").

Now a **`cache Index_Load_Tag_D, KSEG0_BASE+0`**[19] instruction fetches half the configuration information into *DTagLo*, and a **`cache Index_Load_Tag, KSEG0_BASE+8`** gets the other half (the "8" steps to the next feasible tag location - an artefact of the 64-bit width of the cache interface.) The corresponding operations directed at the primary I-cache read the halves of the I-side scratchpad tag, this time into *ITagLo*. The "tag" for I-side and D-side SPRAM appears in *TagLo* fields as shown in Figure 5.8.

[18] What follows is a hardware convention which SoC designers are not compelled to follow; but MIPS Technologies recommends designers to do SPRAM this way to ease software porting.

[19] The instructions are written as if using C "`#define`" names from [m32c0 h]

Figure 5.8 SPRAM (scratchpad RAM) configuration information in TagLo

	31 12	11 8	7	6 5 4 1 0
addr == 0	base address[31:12]	0	En	0
addr == 8	size of region in bytes/4KB	0	En	0

Where:

- *base address[31:12]*: the high-order bits of the physical base address of this chunk of SPRAM;
- *En*: enable the SPRAM. From power-up this bit is zero, and until you set it to 1 the SPRAM is invisible. The *En* bit is also visible in the second (size) configuration word — it can even be written there, but it's not a good idea to write the size word other than for far-out diagnostics;
- *size of region in bytes/4KB*: the number of page-size chunks of data mapped. If you take the whole 32 bits, it returns the size in bytes (but it will always be a multiple of 4KB).

In some MIPS CPUs using this sort of tag setup there could be multiple scratchpad regions indicated by two or more of these tag pairs. But the reference design provided with the 1004K CPU can only have one I-side and one D-side region.

You can load software into the ISPRAM using cacheops. Each pair of instructions to be loaded are put in the registers *IDataHi*/*IDataLo*, and then you use a **`cache Index_Store_Data_I`** at the appropriate index. The two data registers work together to do a 64-bit transfer. For a CPU configured big-endian the first instruction in sequence is loaded into *IDataHi*, but for a CPU configured little-endian the first instruction is loaded into *IDataLo*.

Don't forget to set *ErrCtl[SPR]* back to zero when you're done.

5.7 Common Device Memory Map

In order to preserve the limited CP0 register address space, many new architectural enhancements, particularly those requiring several registers, will be memory mapped, that is, accessed by uncached load and store instructions. In order to avoid creating dozens of memory regions to be managed, the common device memory map (CDMM) was created to group them into one region. A single physical address region, up to 32KB, is defined for CDMM. The address of this region is programmable via the *CDMMBase* CP0 register shown in Figure 5-9.

Having this region physically addressed enables some additional access controls. On a CPU with a TLB, the region would typically be located in the first 256MB, allowing direct kseg1 access. However, if user or supervisor access was desired, TLB mappings could be established to map a useg address to the same region. On FMT based CPUs, it might be mapped to a kseg1 address if user access was not needed, or to a useg/kuseg address if it was.

The block of addresses is further broken up into 64-byte Device Register Blocks(DRB). A 'device' (feature requiring memory mapped accesses), can use from 1-63 DRBs - up to 4KB of addressable registers. The first 64 bits of the first DRB associated with a device is allocated for an Access Control and Status Register (of which only 32 are in use currently). The ACSR provides information about the device - ID, version, and size - and also contains control bits that can enable user and supervisor read and/or write access to the device. This register is shown in Figure 5.10

CDMM devices are packed into the lowest available DRBs. Starting with 0 (or 1 if *CDMMBase[CI]* ==1), software should read the ACSR, determining both the current device type as well as the starting location for the next device.

Iterating through this process will create a map of all devices which you would presumably store in a more convienent format.

The first device that has been defined in CDMM is the Fast Debug Channel which is described in Section 10.1.11 "Fast Debug Channel". This device is a UART-like communication channel that utilizes the EJTAG pins for off-chip access. The UART is a natural fit for a memory mapped device, although many types of devices can be envisioned.

Figure 5-9 Fields in the CDMMBase Register

31 11	10	9	8 0
CDMM_UPPER_ADDR	EN	CI	CDMMSize
	0		

Where:

CDMM_UPPER_ADDR:: This field contains the upper bits of the base physical address of the CDMM region. This field is shifted by 4b, so that bits 31..11 correspond to PA bits 35..15. Unimplemented physical address bits such as 35..32 in many CPUs will be tied to 0.

EN: Enables CDMM. When this bit is cleared, loads and stores to the CDMM region will go to memory. This bit resets to 0 to avoid stepping on other things in the system address map.

CI: Indicates that the first 64-byte device register block is reserved for additional CDMM information and is not a normal device. This extra information hasn't been dreamed up yet, so this field should just be treated as reserved.

CDMMSize: This field indicates how many 64-byte device register blocks are in the CDMM region. (0 means 1 DRB and so forth)

Each device within the CDMM begins with an Access Control and Status Register which gives information about the device and also provides a means for giving user and supervisor programs access to the rest of the device. The *FDACSR* is shown in Figure 5.10.

Figure 5.10 Fields in the Access Control and Status (ACSR) Register

31 24	23 22	21 16	15 12	11 4	3	2	1	0
DevID	zero	DevSize	DevRev	zero	Uw	Ur	Sw	Sr

Where:

DevID: (read only) indicates the device ID.

DevSize: (read only) indicates how many additional 64B blocks this device uses

DevRev: (read only) Revision number of the device.

Uw/Ur: control whether write and reads, respectively, from user programs are allowed to access the device registers. If 0, reads will return 0 and writes will be dropped.

Sw/Sr: Same idea as *Uw/Ur*, but for supervisor access

5.8 The TLB and translation

The TLB is the key piece of hardware which MIPS architecture CPUs have for memory management. It’s a hardware array, and for maintenance you access fields by their index. For memory translation, it’s a real content-addressed memory, whose input is a virtual page address together with the “address space identifier” from *EntryHi[ASID].* The table also stores a physical address plus “cacheability” attributes, which becomes the output of the translation lookup.

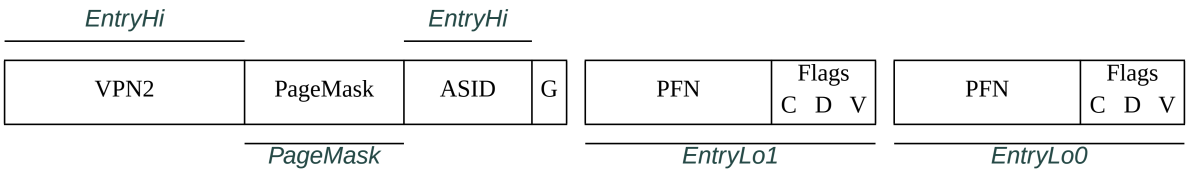
The hardware TLB is relatively small, configurable with 16, 32 or 64 entries (read *Config1[MMUSize]* for the number configured for your CPU). Each entry can map a 2-page-size virtual region to a pair of physical pages. Entries can map different size pages, too.

System software maintains the TLB as a cache of a much larger number of possible translations. An attempt to use a mapped-space address for which no translation is in the hardware TLB invokes a special exception handler which is carefully crafted to find and load the right entry as quickly as possible. Read on for a summary of all the fields and how it gets used; but the OS ramifications are far too extensive to cover here; for a better description in context see [SEEMIPSRUN]:, and for full details of the architectural specification see [MIPS32].

5.8.1 A TLB entry

Let’s start with a sketch of a TLB entry. For MIPS32 CPUs, that consists of a virtual address portion to match against and two output sections, something like Figure 5.11 - which also shows which TLB fields are carried in which CP0 registers.

Figure 5.11 Fields in a 1004K™ CPU TLB entry



Some points to make about the TLB entry:

- The input-side virtual address fields (to the left) have the fields necessary to match an incoming address against this entry. “VPN” is (by OS tradition) a “virtual page number” - the high bits of the program (virtual) address.

  “VPN2” is used to remind you that this address is for a double-page-size virtual region which will map to a pair of physical pages...

- The right-hand side (physical) fields are the information used to output a translation. There are a pair of outputs for each input-match, and which of them is used is determined by the highest within-match address bit. So in standard form (when we’re using 4Kbyte pages) each entry translates an 8Kbyte region of virtual address, but we can map each 4Kbyte page onto any physical address (with any permission flag bits).

- The size of the input region is configurable because the “PageMask” determines how many incoming address bits to match. The 1004K CPU allows page sizes of 4Kbytes, 16Kbytes and going on in powers of 4 up to 256Mbytes. That’s expressed by the legal values of *PageMask*, shown below.

- The “ASID” field extends the virtual address with an 8-bit, OS-assigned memory-space identifier so that translations for multiple different applications can co-exist in the TLB (in Linux, for example, each application has different code and data lying in the same virtual address region).

- The “G” (global) bit is not quite sure whether it’s on the input or output side - there’s only one, but it can be read and written through either of *EntryLo0-1*. When set, it causes addresses to match regardless of their ASID value, thus defining a part of the address space which will be shared by all applications. For example, Linux applications share some “kseg2” space used for kernel extensions.

5.8.2 The TLB and Multithreading

CPUs in the 1004K family are built with just one piece of TLB hardware. However, you can configure your CPU with the TLB either shared between two VPEs, or partitioned so that each VPE sees a standard (though smaller) TLB array.

TLB sharing will usually provide the best performance for when the VPEs are running the same kernel, are closely collaborating, or when one of them makes little or no use of translated addresses. TLB sharing is not completely software-transparent, and some OS work will be needed. See Section 4.3.4 “Sharing and not sharing the TLB” for details.

5.8.3 Live translation and micro-TLBs

When you’re really tuning out the last cycle, you need to know that in the 1004K CPU the translation is actually done by two little tables local to the instruction fetch unit and the load/store unit - called the ITLB and DTLB respectively (collectively they’re “micro-TLBs” or “uTLBs”). There are only 4 entries in the ITLB, and 8 in the DTLB and they are functionally invisible to software: they’re automatically refilled from the main TLB (in this context it’s often called the *joint TLB* or *JTLB*) when required, and automatically cleared whenever the TLB is updated. It costs just three extra clocks to refill the uTLB for any access whose translation is not already in the appropriate uTLB. uTLB entries can only map 4KB and 1MB pages (main TLB entries can handle a whole range of sizes from 4KB to 256MB). When the uTLB is reloaded a translation marked for a size other than 4KB or 1MB is down-converted as required.

5.8.4 Reading and writing TLB entries: Index, Random and Wired

Two CP0 registers work as simple indexes into the TLB array for programming: *Index* and *Random*. The oddly-named *Wired* controls *Random*’s behavior.

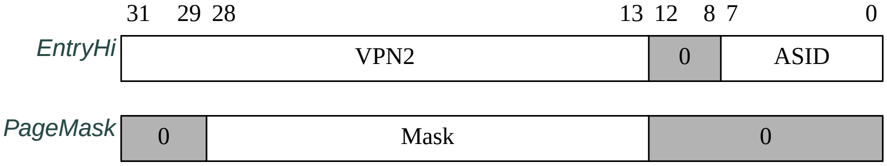
Of these: *Index* determines which TLB entry is accessed by **`tlbwi`**. It’s also used for the result of a **`tlbp`** (the instruction you use to see whether a particular address would be successfully translated by the CPU). *Index* only implements enough bits to index the TLB, however big that is; but a **`tlbp`** which fails to find a match for the specified virtual address sets bit 31 of *Index* (it’s easy to test for).

Random is implemented as a full CPU clock-rate downcounter. It won’t decrement below the value of *Wired* (when it gets there it bounces off and starts again at the highest legal index). In practice, when used inside the TLB refill exception handler, it delivers a random index into the TLB somewhere between the value of *Wired* and the top. *Wired* can therefore be set to reserve some TLB entries from random replacement - a good place for an OS to keep translations which must never cause a TLB translation-not-present exception. In a MT CPU like 1004K, Random is a per-VPE register; and to avoid the danger that random replacement of a TLB entry inside the TLB miss exception handler could conflict with programmed updates made by the other VPE Random is prevented from taking the same value as the other VPE’s Index. See Section 4.3.4 “Sharing and not sharing the TLB” for more on this unexpected feature.

5.8.5 Reading and writing TLB entries - EntryLo0-1, EntryHi and PageMask registers

The TLB is accessed through staging registers which between them represent all the fields in each TLB entry; they're called *EntryHi, PageMask* and *EntryLo0-1*. The fields from *EntryHi* and *PageMask* are shown in Figure 5.12.

Figure 5.12 Fields in the EntryHi and PageMask registers



All these fields act as staging posts for entries being written to or read from the TLB. But some of them are more magic than that...

EntryHi[VPN2]: is the page-pair address to be matched by the entry this reads/writes - see above.

However, on a TLB-related exception *VPN2* is automagically set to the virtual address we were trying to translate when we got the exception. If - as is most often the case - the outcome of the exception handler is to find and install a translation to that address, *VPN2* (and generally the whole of *EntryHi*) will turn out to already have the right values in it.

EntryHi[ASID]: does double-duty. It is used to stage data to and from the TLB, but in normal running software it's also the source of the current "ASID" value, used to extend the virtual address to make sure you only get translations for the current process. Because of that role it is replicated per-TC in MIPS MT systems, and is also visible as *TCStatus[TASID]*.

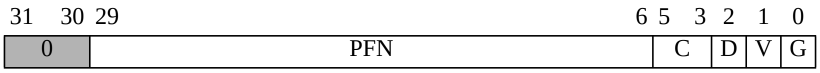
PageMask[Mask]: acts as a kind of backward mask, in that a 1 bit means "don't compare this address bit when matching this address". However, only a restricted range of *PageMask* values are legal (that's with "1"s filling the *PageMask[Mask]* field from low bits upward, two at a time):

`PageMask`	*Size of each output page*	`PageMask`	*Size of each output page*
`0x0000.0000`	4Kbytes	`0x007F.E000`	4Mbytes
`0x0000.6000`	16Kbytes	`0x01FF.E000`	16Mbytes
`0x0001.E000`	64Kbytes	`0x07FF.E000`	64Mbytes
`0x0007.E000`	256Kbytes	`0x1FFF.E000`	256Mbytes
`0x001F.E000`	1Mbyte		

Note that the uTLBs handle only 4Kbyte and 1MB page sizes; other page sizes are down-converted to 4Kbyte or 1MB as they are referenced. For other page sizes this may cause an unexpectedly high rate of uTLB misses, which could be noticeable in unusual circumstances.

Then moving our attention to the output side, the two *EntryLo0-1* are identical in format as shown in Figure 5.13.

Figure 5.13 Fields in the EntryLo0-1 registers



In *EntryLo0-1*:

PFN: the "physical frame number" - traditional OS name for the high-order bits of the physical address. 24 bits of *PFN* together with 12 bits of in-page address make up a 36-bit physical address; but the 1004K CPU has a 32-bit physical address bus, and does not implement the four highest bits (which always read back as zero).

C: a code indicating how to cache data in this page - pages can be marked uncacheable and various flavours of cacheable. The codes here are shared with those used in CP0 registers for the cacheability of fixed address regions: see Table 5.3 in Section 5.4.2, "Cacheability options" on page 74 .

D: the "dirty" flag. In hardware terms it's just a write-enable (when it's 0 you can't do a store using addresses translated here, you'll get an exception instead). However, software can use it to track pages which have been written to; when you first map a page you leave this bit clear, and then a first write causes an exception which you note somewhere in the OS' memory management tables (and of course remember to set the bit).

V: the "valid" flag. You'd think it doesn't make much sense - why load an entry if it's not valid? But this is very helpful so you can make just one of a pair of pages valid.

G: the "global" bit. This really belongs to the input side, and there aren't really two independent values for it. So you should always make sure you set *EntryLo0[G]* and *EntryLo1[G}* the same.

5.8.6 TLB initialization and duplicate entries

TLB entries come up to random values on power-up, and must be initialized by hardware before use. Generally, early bootstrap software should go through setting each entry to a harmless "invalid" value.

Since the TLB is a fully-associative array and entries are written by index, it's possible to load duplicate entries - two or more entries which match the same virtual address/ASID. In older MIPS CPUs it was essential to avoid duplicate entries - even duplicate entries where all the entries are marked "invalid". Some designs could even suffer hardware damage from duplicates. Because of the need to avoid duplicates, even initialization code ought to use a different virtual address for each invalid entry; it's common practice to use "kseg0" virtual addresses for the initial all-invalid entries.

Most MIPS Technologies CPUs protect themselves and you by taking a "machine check" exception if a TLB update would have created a duplicate entry.But the attempted creation of duplicate entries is difficult to prevent on a multi-threaded CPU.If you're using more than one TC in the same address space in a memory-mapped system, there's a chance that two threads will access the same currently-unmapped page, so that both threads cause a TLB refill exception, and the second one will try to load a valid duplicate mapping. In the 1004K CPU, no machine check exception is taken but rather the hardware detects the duplicate valid mapping and simply doesn't do the second write. While this particular example would generally utilize the tlbwr instruction, the hardware will drop both tlbwi and tlbwr writes when a conflict is detected.

More recent (non-MT) CPUs only only take a machine check exception if both of the conflicting entries are valid. Some earlier MIPS Technologies CPUs suffer a machine check even if duplicate entries are both invalid. That can happen when initializing. For example, when an OS is initializing the TLB it may well re-use the same entries as already exist - perhaps the ROM monitor already initialized the TLB, and (derived from the same source code) happened to use the same dummy addresses. If you do that, your second initialization run will cause a machine check exception. The solution is for the initializing routine to check the TLB for a matching entry (using the **`tlbp`** instruction) before each update.

For portability you should probably include the probe step in initialization routines: it's not essential on the 1004K CPU or any machine conforming to the MIPS MT ASE, where we repeat that the machine check exception doesn't happen.

5.8.7 TLB exception handlers — BadVaddr, Context, and ContextConfig registers

These three registers are provided mainly to simplify TLB refill handlers.

BadVAddr is a plain 32-bit register which holds the virtual address which caused the last address-related exception, and is read-only. It is set for the following exception types only: Address error (AdEL or AdES), TLB/XTLB Refill, TLB Invalid (TLBL, TLBS) and TLB Modified (for more on exception codes in *Cause[ExcCode]*, see the notes to Table C.4.)

Context contains the useful mix of pre-programmed and borrowed-from-*BadVAddr* bits shown in Figure 5.14.

Figure 5.14 Fields in the Context register when Config3$_{CTXTC}$=0 and Config3$_{SM}$=0

31 23	22 4	3 0
PTEBase	BadVPN2	0

Context[PTEBase,BadVPN2]: the *PTEBase* field is just software-writable and readable, with no hardware effect.

The *PTEBase* field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer into the current PTE array in memory. The field has no direct hardware effect. The *BadVPN2* field is written by hardware on a TLB exception. It contains bits $VA_{31..13}$ of the virtual address that caused the exception.

In a preferred scheme for software management of page tables, *PTEBase* can be set to the base address of a (suitably aligned) page table in memory; then the *BadVPN2* number (see below) comes from the virtual address associated with the exception—-it's just bits from *BadVAddr*, repackaged. In this case the virtual address bits are shifted such that each ascending 8Kbyte translation unit generates another step through a page table (assuming that each entry is 2 x 32-bit words in size — reasonable since you need to store at least the two candidate *EntryLo0-1* values in it).

An OS which can accept a page table in this format can contrive that in the time-critical simple TLB refill exception, *Context* automagically points to the right page table entry for the new translation.

This is a great idea, but modern OS' tend not to use it — the demands of portability mean it's too much of a stretch to bend the page table information to fit this model.

If *Config3*$_{CTXTC}$ =0 and *Config3*$_{SM}$ =0, then the *Context* register is organized in such a way that the operating system can directly reference a 16-byte structure in memory that describes the mapping. For PTE structures of other sizes, the content of this register can be used by the TLB refill handler after appropriate shifting and masking.

If *Config3*$_{CTXTC}$ =0 and *Config3*$_{SM}$ =0 then a TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31..13}$ of the virtual address to be written into the *BadVPN2* field of the *Context* register. The *PTEBase* field is written and used by the operating system.

The *BadVPN2* field of the *Context* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence.

Figure 5.14 shows the format of the *Context* Register when *Config3*$_{CTXTC}$ =0 and *Config3*$_{SM}$ =0.

If *Config3*$_{CTXTC}$ =1 or *Config3*$_{SM}$ =1 then the pointer implemented by the *Context* register can point to any power-of-two-sized PTE structure within memory. This allows the TLB refill handler to use the pointer without additional shifting and masking steps. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

If *Config3*$_{CTXTC}$ =1 or *Config3*$_{SM}$ =1 then the a TLB exception (Refill, Invalid, or Modified) causes bits $VA_{X+9:Y+9}$ to be written to a variable range of bits "(X-1):Y" of the *Context* register, where this range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 31:X are R/W to software, and are unaffected by the exception. Bits Y-1:0 will always read as 0. If X = 23 and Y = 4, i.e. bits 22:4 are set in *ContextConfig*, the behavior is identical to the standard MIPS32 *Context* register (bits 22:4 are filled with $VA_{31:13}$). Although the fields have been made variable in size and interpretation, the MIPS32 nomenclature is retained. Bits 31:X are referred to as the *PTEBase* field, and bits X-1:Y are referred to as *BadVPN2*.

The value of the *Context* register is **UNPREDICTABLE** following a modification of the contents of the *ContextConfig* register.

Figure 5.15 shows the format of the *Context* Register when *Config3*$_{CTXTC}$ =1 or *Config3*$_{SM}$ =1.

Figure 5.15 Fields in the Context register when Config3$_{CTXTC}$=1 or Config3$_{SM}$=1

31 X	X-1 Y	Y-1 0
PTEBase	BadVPN2	0

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected of the *Context* register are R/W to software and serve as the *PTEBase* field. Bits below the selected field of the *Context* register will read as zeroes.

The field to contain the virtual address index is defined by a single block of contiguous non-zero bits within the *ContextConfig* register's *VirtualIndex* field. Any zero bits to the right of the least significant one bit cause the corresponding *Context* register bits to read as zero. Any zero bits to the left of the most significant one bit cause the corresponding *Context* register bits to be R/W to software and unaffected by TLB exceptions.

A value of all ones in the *ContextConfig* register means that the full 32 bits of the faulting virtual address will be copied into the context register, making it duplicate the *BadVAddr* register. A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

The *ContextConfig* register is optional and its existence is denoted by the *Config3*$_{CTXTC}$ or *Config3*$_{SM}$ register fields.

Figure 5.16 shows the formats of the *ContextConfig* Register.

Figure 5.16 Fields in the ContextConfig register

31 0
VirtualIndex

VirtualIndex is a mask of 0 to 32 contiguous 1 bits that cause the corresponding bits of the *Context* register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is **UNDEFINED** if non-contiguous 1 bits are written into the register field.

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. It is possible for software to determine which bits are implemented by alternately writing all zeroes and all ones to the register, and reading back the resulting values. Table 5.8 describes some useful *ContextConfig* values.

Table 5.8 Recommended ContextConfig Values

Value	Page Table Organization	Page Size	PTE Size	Compliance
0x00000000007ffff0	Single Level	4K	64 bits/page	REQUIRED
0x00000000003ffff8	Single Level	4K	32 bits/page	RECOMMENDED
0x00000000007ffff8	Single Level	2K	32 bits/page	RECOMMENDED
0x0000000000fffff8	Single Level	1K	32 bits/page	RECOMMENDED

Chapter 6

Programming the 1004K™ CPU in user mode

This chapter is not very long, because in user mode one MIPS32-compliant CPU looks much like another. But not everything — sections include:

- Section 6.1, "User-mode accessible “Hardware registers”"
- Section 6.2, "Prefetching data": how it works.
- Section 6.3, "Using “synci” when writing instructions": writing instructions without needing to use privileged cache management instructions.
- Section 6.4, "The multiplier": multiply, multiply/accumulate and divide timings.
- Section 6.5, "Tuning software for the 1004K‘ family pipeline": for determined programmers, and for compiler writers. It includes information about the timing of the DSP ASE instructions.
- Section 6.6 “Tuning floating-point”: the floating-point unit often runs at half speed, and some of its interactions (particularly about potential exceptions) are complicated. This section offers some guidance about the timing issues you’ll encounter.

6.1 User-mode accessible “Hardware registers”

The 1004K CPU complies with Revision 2 of the MIPS32 specification, which introduces *hardware registers*; CPU-dependent registers which are readable by unprivileged user space programs, usually to share information which is worth making accessible to programs without the overhead of a system call.

The hardware registers provide useful information about the hardware, even to unprivileged (user-mode) software, and are readable with the **rdhwr** instruction. [MIPS32] defines four registers so far. The OS can control access to each register individually, through a bitmask in the CP0 register *HWREna* - (set bit 0 to enable register 0 etc). *HWREna* is cleared to all-zeroes on reset, so software has to explicitly enable user access — see Section 7.6 “The HWREna register - Control user rdhwr access”. Privileged code can access any hardware register.

The five standard registers are:

- *CPUNum (0)*: Number of the CPU on which the program is currently running. This comes directly from the coprocessor 0 *EBase[CPUNum]* field.
- *SYNCI_Step (1)*: the effective size of an L1 cache line[20]; this is now important to user programs because they can now do things to the caches using the **synci** instruction to make instructions you’ve written visible for execution. Then *SYNCI_Step* tells you the “step size” - the address increment between successive **synci**’s required to cover all the instructions in a range.

[20] Strictly, it’s the lesser of the I-cache and D-cache line size, but it’s most unusual to make them different.

If *SYNCI_Step* returns zero, that means that your hardware ensures that your caches are instruction/data coherent, and you don't need to use **synci** at all.

- *CC (2)*: user-mode read-only access to the CP0 *Count* register, for high-resolution counting. Which wouldn't be much good without.

- *CCRes (3)*: which tells you how fast *Count* counts. It's a divider from the pipeline clock — if the rdhwr instruction reads a value of "2", then *Count* increments every 2 cycles, at half the pipeline clock rate. For 1004K family CPUs that is precisely what you will read.

- *UL (30)*: user-mode read-only access to the CP0 *UserLocal* register. This register can be used to provide a thread identifier to user-mode programs. See Section C.4.2 "The UserLocal register" for more details

6.2 Prefetching data

MIPS32 CPUs are being increasingly used for computations which feature loops accessing large arrays, and the runtime is often dominated by cache misses.

These are excellent candidates for using the **pref** instruction, which gets data into the cache without affecting the CPUs other state. In a well-optimized loop with prefetch, data for the next iteration can be fetched into the cache in parallel with computation for the last iteration.

It's a pretty major principle that **pref** should have *no software-visible effect* other than to make things go faster. **pref** is logically a no-op[21].

The **pref** instruction comes with various possible "hints" which allow the program to express its best guess about the likely fate of the cache line. In 1004K family CPUs the "load" and "store" variants of the hints do the same thing; but it makes good sense to use the hint which matches your program's intention - you might one day port it to a CPU where it makes a difference, and it can't do any harm. Unlike many prior CPU families, in 1004K family CPUs, the "load" and "store" variants of the hints do things differently. Specifically, when dealing with coherent addresses, the "store" types will request Exclusive ownership so that an eventual store does not need to make an upgrade request before modifying the line.

The 1004K CPU acts on hints as summarized in Table 6.1.

6.3 Using "synci" when writing instructions

The **synci** instruction (introduced with Revision 2 of the MIPS32 architecture specification, [MIPS32]) ensures that instructions written by a program (necessarily through the D-cache, if you're running cached) get written back from the D-cache and corresponding I-cache locations invalidated, so that any future execution at the address will reliably execute the new instructions. **synci** takes an address argument, and it takes effect on a whole enclosing cache-line sized piece of memory. User-level programs can discover the cache line size because it's available in a "hardware registers" accessed by **rdhwr**, as described in Section 6.1, "User-mode accessible "Hardware registers"" above.

Since **synci** is modifying the program's own instruction stream, it's inherently an "instruction hazard": so when you've finished writing your instructions and issued the last **synci**, you should then use a **jr.hb** or equivalent to call the new instructions — see Section 7.1 "Hazard barrier instructions".

[21] This isn't quite true any more; **pref** with the "PrepareForStore" hint can zero out some data which wasn't previously zero.

Table 6.1 Hints for “pref” instructions

Hint		*What happens in the 1004K CPU*	*Why would you use it?*
No	*Name*		
0 1	load store	Read the cache line into the D-cache if not present.	When you expect to read the data soon. Use “store” hint if you also expect to modify it.
4 5	load_streamed store_streamed	Fetch data, but always use cache way zero - so a large sequence of “streamed” prefetches will only ever use a quarter of the cache.	For data you expect to process sequentially, and can afford to discard from the cache once processed
6 7	load_retained store_retained	Fetch data, but never use cache way zero. That means if you do a mixture of “streamed” and “retained” operations, they will not displace each other from the cache.	For data you expect to use more than once, and which may be subject to competition from “streamed” data.
25	writeback_invalidate/ nudge	If the line is in the cache, invalidate it (writing it back first if it was dirty). Otherwise do nothing. However (with the 1004K CPU only): if this line is in a region marked for “uncached accelerated write” behavior, then write-back this line.	When you know you’ve finished with the data, and want to make sure it loses in any future competition for cache resources.
30	PrepareForStore	If the line is not in the cache, create a cache line - but instead of reading it from memory, fill it with zeroes and mark it as “dirty”. If the line is already in the cache do nothing - *this operation cannot be relied upon to zero the line*.	When you know you will overwrite the whole line, so reading the old data from memory is unnecessary. A recycled line is zero-filled only because its former contents could have belonged to a sensitive application - allowing them to be visible to the new owner would be a security breach.

6.4 The multiplier

As is traditional with MIPS CPUs, the integer multiplier is a semi-detached unit with its own pipeline. All MIPS32 CPUs implement:

- **mult**/**multu**: a 32×32 multiply of two GPRs (signed and unsigned versions) with a 64-bit result delivered in the multiply unit’s pseudo-registers *hi* and *lo* (readable only using the special instructions **mfhi** and **mflo**, which are interlocked and stall until the result is available).
- **madd**, **maddu**, **msub**, **msubu**: multiply/accumulate instructions collecting their result in *hi*/*lo*.
- **mul**/**mulu**: simple 3-operand multiply as a single instruction.
- **div**/**divu**: divide - the quotient goes into *lo* and the remainder into *hi*.

Many of the most powerful instructions in the MIPS DSP ASE are variants of multiply or multiply-accumulate operations, and are described in Chapter 9, “The MIPS32® DSP ASE” on page 127. The DSP ASE also provides three additional “accumulators” which behave like the *hi*/*lo* pair).

No multiply/divide operation ever produces an exception - even divide-by-zero is silent - so compilers typically insert explicit check code where it’s required.

The 1004K CPU multiplier is high performance and pipelined; multiply/accumulate instructions can run at a rate of 1 per clock, but a 32×32 3-operand multiply takes four clocks longer than a simple ALU operation. Divides use a bit-per-clock algorithm, which is short-cut for smaller dividends. Multiply/divide instructions are generally slow enough that it is difficult to arrange programs so that their results will be ready when needed.

6.5 Tuning software for the 1004K™ family pipeline

This section is addressed to low-level programmers who are tuning software by hand and to those working on efficient compilers or code translators.

Note, though, that when there is a multi-threading workload some of the following issues become less important. There's not so much need to mitigate cache miss delays (for example) when the time when one thread is waiting will be cheerfully used by another thread which keeps running.

The 1004K CPU is a pipelined design, and the pipeline and some of its consequences are described in Section 3.1 "The 1004K™ core pipeline and multithreading" . That leads to a class of possible delays to do with data dependencies and resource limitations.

For software tuning purposes it's usually enough to know the delay which results when one instruction (the "producer") generates a value in some particular register for the use of the next instruction in sequence (the "consumer"). The delay is in processor cycle time units, but it makes good sense to think of that delay as a lost opportunity to run an instruction. To tune round data dependencies, the programmer or compiler needs to re-order the instructions so that enough useful but independent instructions are placed between the producer and consumer that the consumer runs without delay.

There are times when interactions are more complicated than that. While you can pore over hardware books to try to figure out what the pipeline is doing, when it gets that difficult we advise that you should obtain a cycle-accurate simulator or other well-instrumented test environment, and try your software out.

But before getting on to data delays, we'll look at the most important causes of slow-down: cycles lost to cache misses and branches.

6.5.1 Cache delays and mitigating their effect

In a typical 1004K CPU implementation a cache miss which has to be refilled from DRAM memory (in the very next chip on the board) will be delayed by a period of time long enough to run 50-200 instructions. A miss or uncached read (perhaps of a device register) may easily be several times slower. These really are important!

Of course, this is one of the main motivations for having a multithreading CPU: while one thread is stopped because of a cache miss, other threads can keep running, greatly improving the total throughput.

Because these delays are so large, there's not a lot you can do to help a cache-missing program make progress. But every little bit helps. The 1004K CPU has non-blocking loads, so if you can move your load instruction producer away from its consumer, you won't start paying for your memory delay until you try to run the consuming instruction.

Compilers and programmers find it difficult to move fragments of algorithm backwards like this, so the architecture also provides prefetch instructions (which fetch designated data into the D-cache, but do nothing else). Because they're free of most side-effects it's easier to issue prefetches very early. Any loop which walks predictably through a large array is a candidate for prefetch instructions, which are conveniently placed within one iteration to prefetch data for the next.

The **`pref PrepareForStore`** prefetch saves a cache refill read, for cache lines which you intend to overwrite in their entirety. Read more about prefetch in Section 6.2, "Prefetching data" above.

Tuning data-intensive common functions

Bulk operations like `bcopy()` and `bzero()` will benefit from CPU-specific tuning. To get excellent performance for in-cache data, it's only necessary to reorganize the software enough to cover the address-to-store and load-to-use delays. But to get the loop to achieve the best performance when cache missing, you probably want to use some prefetches. MIPS Technologies may have example code of such functions — ask.

6.5.2 Branch delay slot

It's a feature of the MIPS architecture that it always attempts to execute the instruction immediately following a branch. The rationale for this is that it's extremely difficult to fetch the branch target quickly enough to avoid a delay, so the extra instruction runs "for free"...

Most of the time, the compiler deals well with this single delay slot. MIPS low-level programmers find it odd at first, but you get used to it!

6.6 Tuning floating-point

It seemed to make more sense to put this information into the FPU chapter: read from Section 8.5 "FPU pipeline and instruction timing".

6.6.1 Branch misprediction delays

In a long-pipeline design like this, branches would be expensive if you waited until the branch was executed before fetching any more instructions. See Section 3.1, "The 1004K™ core pipeline and multithreading" for what is done about this: but the upshot is that where the fetch logic can't compute the target address, or guesses wrong, that's going to cost five or more lost cycles. It does depend what sort of branch: the conditional branch which closes a tight loop will almost always be predicted correctly after the first time around. Moreover, in an MT CPU like the 1004K core most of the pain is felt by the thread which executes the branch; so long as there are other running threads the CPU can keep busy.

However, too many branches in too short a period of time can overwhelm the ability of the instruction fetch logic to keep ahead with its predictions. Where branchy code can be replaced by conditional moves, you'll get significant benefits.

The branch-likely[22] instructions (officially deprecated by the MIPS32 architecture because they may perform poorly on more sophisticated or wider-issue hardware) are predicted just like any other branch.

Although deprecated, the branch-likely instructions will probably improve the performance of loops where there is no other way of avoiding a no-op in a loop-closing branch's delay slot. If you're tempted to use this, we strongly recommend you make the code conditional on a `#define` variable tied specifically to the 1004K family. If that's difficult in your environment and the code might need to be portable, it's probably better not to use this.

6.6.2 Data dependency delays classified

We've attempted to tabulate all possible producer/consumer delays affecting user-level code (we're not discussing CP0 registers here), but excluding floating point (which is in the next section).

In fact, we won't set out the tables exactly like that. The MIPS instruction set is efficient because, most of the time, dependent instructions can be run nose-to-tail without delay. For all registers, there is a "standard" place in the pipeline where the producer should deliver its value and another place in the pipeline where the consumer picks it up[23]. Producer/consumer delays happen when either the producer is late delivering a result to the register (we'll abbreviate to "lazy"), or the consumer insists on obtaining its operand early (we'll abbreviate to "eager"). Of course, both may happen: in that case the delays add up.

It's important to be clear what class of registers is involved in any of these delays. For non-floating-point user-level code, there are just three classes of registers to consider:

- General purpose registers ("GPR");
- The *hi/lo* pair together with the three additional accumulators defined by the MIP DSP ASE "accumulator" pair ("ACC");
- The fields of the *DSPControl* register.

So that gives us two tables.

[22] The "likely" in the instruction name is historical, and pretty misleading.
[23] These are brought closer together by the magic of register file bypasses, but we don't need to get into the details here.

Delays caused by “eager consumers” reading values early

Table 6.2 Register → eager consumer delays

Reg	→	*Eager consumer*	*Del*	*Applies when...*
GPR	→	load/store	1	the GPR value is an address operand (store data is not needed early).
ACC	→	multiply instructions	1	the ACC value came from any *non-multiply* or *multiply instructions which saturate the accumulator value* (values generated by other multiply instructions are made available early, and thus avoid this delay).
ACC	→	DSP instructions which extract selected bits from an accumulator: **`extp`**..., **`extr`**... etc.	3	Always
		DSP instructions which write a shifted value back to the accumulator: **`mthlip`**, **`shilo`**, **`shilov`**.		

Delays caused by “lazy producers” delivering values late

Table 6.3 Lazy producer → register delays

Lazy producer	→	*Reg*	*Del*	*Applies when...*
Load	→	GPR	1	Always (familiar as the “load delay slot”).
Integer multiply unit instructions producing a GPR result.	→	GPR	4	Always (because the multiply unit pipeline is longer than the integer unit’s).
Instructions reading accumulators and writing GPR (e.g. **`mflo`**).				
DSP “ALU” instructions (which neither read nor write an accumulator, nor do a multiplication).	→	GPR	1	Always
Integer divide instruction	→	ACC	7	8-bit dividend
			9	8-bit dividend & negative operand to **`div`**
			15	16-bit dividend
			17	16-bit dividend & negative operand to **`div`**
			23	24-bit dividend
			25	24-bit dividend & negative operand to **`div`**
			31	full-size dividend
			33	full-size dividend & negative operand to **`div`**

How to use the tables

Suppose we’ve got an instruction sequence like this one:

```
addiu     $a0, $a0, 8
lw        $t0, 0($a0)   # [1]
lw        $t1, 4($a0)
addu      $t2, $t0, $t1# [2]
mul       $v0, $t2, $t3
sw        $v0, 0($a1)   # [3]
```

Then a look at the tables should help us discover whether any instructions will be held up. Look at the dependencies where an instruction is dependent on its predecessor:

[1] The **`lw`** will be held up by one clock, because its GPR address operand **`$a0`** was computed by the immediately preceding instruction (see the first box of Table 6.2.) The second **`lw`** will be OK.

[2] The **`addu`** will be one clock late, because the load data from the preceding **`lw`** arrives late in the GPR **`$t1`** (see the first box of Table 6.3.)

[3] The **`sw`** will be 4 clocks late starting while it waits for a result from the multiply pipe (the second box of Table 6.3.)

These can be additive. In the pointer-chasing sequence:

```
lw          $t1, 0($t0)
lw          $t2, 0($t1)
```

The second load will be held up two clocks: one because of the late delivery of load data in **`$t1`** (first box of Table 6.3), plus another because that data is required to form the address (first box of Table 6.2.)

Delays caused by dependencies on DSPControl fields

Some DSP ASE instructions are dependent because they produce and consume values kept in fields of the *DSPControl* register. However, the most performance-critical of these dependencies are “by-passed” to make sure no delay will occur - those are the dependencies between:

```
addsc  →       DSPControl[c]        →  addwc
cmp.x  →     DSPControl[ccond]      →  pick.x
wrdsp  →  DSPControl[pos,scount]    →  insv
```

But other dependencies passed in *DSPControl* may cause delays; in particular the *DSPControl[ouflag]* bits set by various kinds of overflow are not ready for a succeeding **`rddsp`** instruction. The access is interlocked, and will lead to a delay of up to three clocks. We don’t expect that to be a problem (but if you know different, please get in touch with MIPS Technologies).

More complicated dependencies

There can be delays which are dependent on the dynamic allocation of resources inside the CPU. In general you can’t really figure out how much these matter by doing a static code analysis, and we earnestly advise you to get some kind of high-visibility cycle-accurate simulator or trace equipment (probably based on Section 10.2, "PDtrace™ instruction trace facility").

Advice on tuning DSP ASE instruction sequences

DSP algorithm functions are often the subject of intense tuning. There is more specific and helpful advice (with examples) included in the white paper [DSPWP] published by MIPS Technologies.

Chapter 7

Kernel-mode (OS) programming and Release 2 of the MIPS32® Architecture

[MIPS32] tells you how to write OS code which is portable across all compliant CPUs. Most OS code should be CPU-independent, and we won't tell you how to write it here. But release 2 of the MIPS32 Specification [MIPS32] introduced a few new optional features which are not yet well known, so are worth describing here:

- A better way of managing software-visible pipeline and hardware delays associated with CP0 programming in Section 7.1, "Hazard barrier instructions".
- New interrupt facilities described in Section 7.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)";
- That led on to Section 7.3.1 "Summary of exception entry points": where do exceptions go, and what options are available?
- The ability to use one or more extra sets of registers ("shadow sets") to reduce context-saving overhead in interrupt handlers, in Section 7.4, "Shadow registers".
- How to get at any power-saving features, in Section 7.5, "Saving Power"
- How to control user-privilege access to "hardware registers", in Section 7.6 "The HWREna register - Control user rdhwr access".

7.1 Hazard barrier instructions

When privileged "CP0" instructions change the machine state, you can get unexpected behavior if an effect is deferred out of its normal instruction sequence. But that can happen because the relevant control register only gets written some way down the pipeline, or because the changes it makes are sensed by other instructions early in their pipeline sequence: this is called a CP0 *hazard*.

It's possible to get hazards in user mode code too, and many of the instructions described here are not solely for kernel-privilege code. But they're most often met around CP0 read/writes, so they found their way to this chapter.

Traditionally, MIPS CPUs left the kernel/low-level software engineer with the job of designing sequences which are guaranteed to run correctly, usually by padding the dangerous operation with enough **`nop`** or **`ssnop`** instructions.

From Release 2 of the MIPS32 specification this is replaced by explicit *hazard barrier* instructions. If you execute a hazard barrier between the instruction which makes the change (the "producer") and the instruction which is sensitive to it (the "consumer"), you are guaranteed that the change will be seen as complete. Hazards can appear when the producer affects even the instruction fetch of the consumer - that's an "instruction hazard" - or only affecting the operation of the consuming instruction (an "execution hazard"). Hazard barriers come in two strengths: **`ehb`** deals only with execution hazards, while **`eret`**, **`jr.hb`** and **`jalr.hb`** are barriers to both kinds of hazard.

In most implementations the strong hazard barrier instructions are quite costly, often discarding most or all of the pipeline contents: they should not be used indiscriminately. For efficiency you should use the weaker **ehb** where it is enough. Since some implementations work by holding up execution of all instructions after the barrier, it's preferable to place the barrier just before the consumer, not just after the producer.

For example you might be updating a TLB entry:

```
mtc0 Index, t0
# other stuff, if there's stuff to do
ehb
tlbwi
jr.hb ra
```

The **ehb** makes sure that the change to *Index* has been made before you attempt to write the TLB entry, which is fine. But updating the TLB might affect how instructions are fetched in mapped space, so you should not return to code which might be running in mapped space until you've cleared the "instruction hazard". That's dealt with by the **jr.hb**.

Hazard barriers and multi-threading

Within a thread the hazard barriers work as advertised. But because TCs share many CP0 registers and other resources, some hazards can be between different threads - or more precisely, an instruction can produce some effect on other threads which affect the behavior of subsequent instructions.

In particular, the operations which disable other threads (instructions like **dmt** or **dvpe** or direct manipulation of the associated CP0 bits *VPECtl[TE]* and *MVPCtl[EVP]*. or writes to *TCHalt*) may not be immediate. Instructions after the other-thread-disable instruction in the stream might - according to the MT ASE specification [MIPSMT] - see evidence of other threads continuing to run for a while. The MT ASE defines this as an instruction hazard. However, no hazard of this kind exists in 1004K family CPUs, so if you're prepared to make your software CPU-dependent you may make it a bit more efficient.

Porting software to use the new instructions

If you know your software will only ever run on a MIPS32 Release 2 or higher CPU, then that's great. But to maintain software which has to continue running on older CPUs:

- *ehb is a no-op*: on all previous CPUs. So you can substitute an **ehb** for the last no-op in your sequence of "enough no-ops", and your software is now safe on all future CPUs which are compliant with Release 2.

- *jr.hb and jalr.hb*: are decoded as plain jump-register and call-by-register instructions on earlier CPUs. Again, provided you already had enough no-ops for your worst-case older CPU, your system should now be safe on Release 2 and higher CPUs.

7.2 MIPS32® Architecture Release 2 - enhanced interrupt system(s)

The features for handling interrupts include:

- Vectored Interrupt (VI) mode offers multiple entry points (one for each of the interrupt sources), instead of the single general exception entry point.

  External Interrupt Controller (EIC) mode goes further, and reinterprets the six CPU interrupt input signals as a 64-value field - potentially 63 distinguished interrupts each with their own entry point (the zero code, of course, is reserved to mean "no interrupt active").

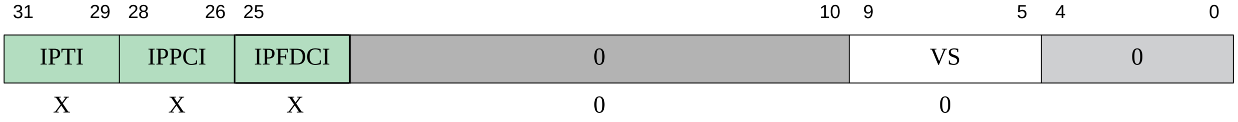
Both these modes need to be explicitly enabled by setting bits in the *Config3* register; if you don't do that, the CPU behaves just as the original (release 1) MIPS32 specification required.

- Shadow registers - alternate sets of registers, often reserved for interrupt handlers, are described in Section 7.4, "Shadow registers". Interrupt handlers using shadow registers avoid the overhead of saving and restoring user GPR values.

- The *Cause[TI]*, *Cause[FDCI]*,and *Cause[PCI]* bits (see the notes to Figure C.2) provide a direct indication of pending interrupts from the on-CPU timer, fast debug channel, and performance counter subsystems (these interrupts are potentially shared with other interrupt inputs, and it previously required system-specific programming to discover the source of the interrupt and handle it appropriately).

The new interrupt options are enabled by the *IntCtl* register, whose fields are shown in Figure 7.1.

Figure 7.1 Fields in the IntCtl Register



IntCtl[IPTI,IPPCI,IPFDCI]: *IPTI*, *IPPCI*, and *IPFDCI* are read-only 3-bit fields, telling you how internal timer, performance counter, and fast debug channel interrupts are wired up. They are relevant in non-vectored and simple-vectored ("VI") interrupt modes, but not if you're using an EIC interrupt controller.

Read this field to get the number of the *Cause[IPnn]* where the corresponding interrupt is seen. Because *Cause[IP1-0]* are software interrupt bits, unconnected to any input, legal values for *IntCtl[IPTI], IntCtl[IPPCI]* , and *IntCtl[IPFDCI]* are between 2 and 7.

The timer, performance counter, and fast debug channel interrupt signals are taken out to the CPU interface and the SoC designer connects them back to one of the CPU's interrupt inputs. The SoC designer is supposed to hard-wire some CPU inputs which show up as the *IntCtl[IPTI,IPPCI,IPFDCI]* fields to match.

These interrupt outputs are per-VPE, so there are two of them from the 1004K CPU. The *IntCtl* register is also per-VPE, reflecting the local setup.

IntCtl[VS]: is writable to give you software control of the vector spacing; if the value in *VS* is **VS**, you will get a spacing of $32 \times 2^{(VS-1)}$ bytes.

Only values of 1, 2, 4, 8 and 16 work (to give spacings of 32, 64, 128, 256, and 512 bytes respectively). A value of zero gives a zero spacing, so all interrupts arrive at the same address — the legacy behavior.

7.2.1 Traditional MIPS® interrupt signalling and priority

Before we discuss the new features, we should remind you what was there already. On traditional MIPS systems the CPU takes an interrupt exception on any cycle where one of the eight possible interrupt sources visible in *Cause[IP]* is active, enabled by the corresponding enable bit in *Status[IM]*, and not otherwise inhibited. When that happens control is passed to the general exception handler (see Table 7.1 for exception entry point addresses), and is recognized by the "interrupt" value in *Cause[ExcCode]*. All interrupt are equal in the hardware, and the hardware does nothing special if two or more interrupts are active and enabled simultaneously. All priority decisions are down to the software.

Six of the interrupt sources are hardware signals brought into the CPU, while the other two are "software interrupts" taking whatever value is written to them in the *Cause* register.

The original MIPS32 specification adds an option to this. If you set the *Cause[IV]* bit, the same priority-blind interrupt handling happens but control is passed to an interrupt exception entry point which is separate from the general exception handler.

7.2.2 VI mode - multiple entry points, interrupt signalling and priority

The traditional interrupt system fits with a RISC philosophy (it leaves all interrupt priority policy to software). It's also OK with complex operating systems, which commonly have a single piece of code which does the housekeeping associated with interrupts prior to calling an individual device-interrupt handler.

A single entry point doesn't fit so well with embedded systems using very low-level interrupt handlers to perform small near-the-hardware tasks. So Release 2 of the MIPS32 architecture adds "VI interrupt mode" where interrupts are despatched to one of eight possible entry points. To make this happen:

1. *Config3[VInt]* must be set, to indicate that your CPU has the vectored-interrupts feature - but all CPUs in the 1004K family have it;

2. You write *Cause[IV]* = 1 to request that interrupts use the special interrupt entry point; and:

3. You set *IntCtl[VS]* non-zero, setting the spacing between successive interrupt entry points.

Then interrupt exceptions will go to one of eight distinct entry points. The bit-number in *Cause[IP]* corresponding to the highest-numbered active interrupt becomes the "vector number" in the range 0-7. The vector number is multiplied by the "spacing" implied by the OS-written field *IntCtl[VS]* (see above) to generate an offset. This offset is then added to the special interrupt entry point (already an offset of 0x200 from the value defined in *EBase*) to produce the entry point to be used.

If multiple interrupts are active and enabled, the entry point will be the one associated with the higher-numbered interrupt: in VI mode interrupts are no longer all equal, and the hardware now has some role in interrupt "priority".

7.2.3 External Interrupt Controller (EIC) mode

Embedded systems have lots of interrupts, typically far exceeding the six input signals traditionally available. Most systems have an external interrupt controller to allow these interrupts to be masked and selected. If your interrupt controller is "EIC compatible" and you use these features, then you get 63 distinct interrupt entry points.

To do this the same six hardware signals used in traditional and VI modes are redefined as a bus with 64 possible values[24]: 0 means "no interrupt" and 1-63 represent distinct interrupts. That's "EIC interrupt mode", and you're in EIC mode if you would be in VI mode (see previous section) and additionally the *Config3[VEIC]* bit is set. EIC mode is a little deceptive: the programming interface hardly seems to change, but the meaning of fields change quite a bit.

Firstly, once the interrupt bits are grouped the interrupt mask bits in *Status[IM]* can't just be bitwise enables any more. Instead this field (strictly, the 6 high order bits of this field, excluding the mask bits for the software interrupts) is recycled to become a 6-bit *Status[IPL]* ("interrupt priority level") field. Most of the time (when running application code, or even normal kernel code) *Status[IPL]* will be zero; the CPU takes an interrupt exception when the interrupt

[24] The resulting system will be familiar to anyone who's used a Motorola 68000 family device (or further back, a DEC PDP/11 or any of its successors).

controller presents a number higher than the current value of *Status[IPL]* on its “bus” and interrupts are not otherwise inhibited.

As before, the interrupt handler will see the interrupt request number in *Cause[IP]* bits - see Section C.4.3 “Exception handling: Cause register”; the six MS of those bits are now relabelled as *Cause[RIPL]* (“requested IPL”). In EIC mode the software interrupt bits are not used in interrupt selection or prioritization: see below. But there’s an important difference; *Cause[RIPL]* holds a snapshot of the value presented to the CPU when it decided to take the interrupt, whereas the old *Cause[IP]* bits simply reflected the real-time state of the input signals[25].

When an exception is triggered the new IPL - as captured in *Cause[RIPL]* - is used directly as the interrupt number; it’s multiplied by the interrupt spacing implied by *IntCtl[RS]* and added to the special interrupt entry point, as described in the previous section. *Cause[RIPL]* retains its value until the CPU next takes any exception.

Software interrupts: the two bits in *Cause[IP1-0]* are still writable, but now become real signals which are fed out of the CPU CPU, and in most cases will become inputs - presumably low-priority ones - to the EIC-compliant interrupt controller.

In EIC mode the usual association of the internal timer, performance-counter overflow, and fast debug channel interrupts with individual bits of *Cause[IP]* is lost. These interrupts are turned into output signals from the CPU, and will themselves become inputs to the interrupt controller. Ask your system integrator how they are wired.

7.3 Exception Entry Points

Early versions of the MIPS architecture had a rather simple exception system, with a small number of architecture-fixed entry points.

But there were already complications. When a CPU starts up main memory is typically random and the MIPS caches are unusable until initialized; so MIPS CPUs start up in uncached ROM memory space and the exception entry points are all there for a while (in fact, for so long as *Status[BEV]* is set); these “ROM entry points” are clustered near the top of kseg1, corresponding to 0x1FC0.0000 physical[26], which must decode as ROM.

ROM is slow and rigid; handlers for some exceptions are performance-critical, and OS’ want to handle exceptions without relying on ROM code. So once the OS boots up it’s essential to be able to redirect OS-handled exceptions into cached locations mapped to main memory (what exceptions are not OS-handled? well, there are no alternate entry points for system reset, NMI, and EJTAG debug).

So when *Status[BEV]* is flipped to zero, OS-relevant exception entry points are moved to the bottom of kseg0, starting from 0 in the physical map. The cache error exception is an exception... it would be silly to respond to a cache error by transferring control to a cached location, so the cache error entry point is physically close to all the others, but always mapped through the uncached “kseg1” region.

In MIPS CPUs prior to the MIPS32 architecture (with a few infrequent special cases) only common TLB miss exceptions got their own entry point; interrupts and all other OS-handled exceptions were all funneled through a single “general” exception entry point.

[25] Since the incoming IPL can change at any time - depending on the priority views of the interrupt controller - this is essential if the handler is going to know which interrupt it’s servicing.

[26] Even this address can be changed by a brave and determined SoC integrator, see the note on RBASE in Section 7.3.1 “Summary of exception entry points”.

The MIPS32® architecture: interrupts get their own entry point

Embedded systems often make heavy use of interrupts and the OS may be less centralized; so MIPS32 CPUs allow you to redirect all interrupts to a new “special interrupt” entry point; you just set a new bit in the *Cause* register, *Cause[IV]* — see Section C.4.3 “Exception handling: Cause register”

Release 2: relocate all the exception entry points with EBase

The new-with-release 2 *EBase* register does two vital jobs: one is to allow software to know which CPU it’s running on and the other is to relocate the exception entry points. It is primarily supplied for multi-CPU systems (or with a MIPS MT CPU, for systems using multiple VPEs).

The latter is necessary because CPUs sharing a memory map (as SMP CPUs often do and the VPEs inside a MIPS MT CPU are obliged to do) have their exception entry points in kseg0. By setting *EBase* differently on each CPU, you can give them distinct exception handlers.

Figure 7.2 Fields in the EBase Register

31	30	29 12	11 10	9 0
1	0	VA	0	CPUNum
		0		

EBase[VA]: the base address for the exception vectors, adjustable to a resolution of 4Kbytes. See *the exception entry points table* for how that moves all the exception entry points. The top two address bits are fixed to “10”, which means that the base address is constrained to being somewhere in the “unmapped” kseg0/kseg1 memory regions.

By setting *EBase* on any CPU and/or VPE of a multiprocessor and/or multithreading system to a unique value, that CPU can have its own unique exception handlers.

Write this field only when *Status[BEV]* is set, so that any unexpected exception will be handled through the ROM entry points (otherwise you would be changing the exception address under your own feet, and the results of that are undefined).

EBase[CPUNum]: On single-threaded CPUs this is just a single "CPU number" field (set by the CPU interface bus *SI_CPUNum*, which the SoC designer will tie to some suitable value).

7.3.1 Summary of exception entry points

The incremental growth of exception entry points has left no one place where all the entry points are summarized; so here’s Table 7.1. But first:

BASE is 0x8000.0000, as it will be where the software, ignoring the *EBase* register, leaves it at its power-on value — that’s also compatible with older MIPS CPUs. Otherwise BASE is the 4Kbyte-aligned address found in *EBase* after you ignore the low 12 bits...

RBASE is the ROM/reset entry point base, usually `0xBFC0.0000`. However, 1004K family CPUs can be configured to use a different base address by fixing some input signals to the CPU. Specifically, if the CPU is wired with *SI_UseExceptionBase* asserted, then RBASE bits 29-12 will be set by the values of the inputs *SI_ExceptionBase[29:12]* (the two high bits will be “10” to select the kseg0/kseg1 regions, and the low 12 bits are always zero). Relocating RBASE is strictly not compliant with the MIPS32 specification and may break all sorts of useful pieces of software, so it’s not to be done lightly.

DebugVectorAddr is an alternative entry point for debug exceptions. It is specified via a drseg memory mapped register of the same name and enabled through the Debug Control Register. The probe handler still takes precedence, but this is higher priority than the regular ROM entry points.

Table 7.1 All Exception entry points

Memory region	*Entry point*	*Exceptions handled here*
EJTAG probe-mapped	`0xFF20.0200`	EJTAG debug, when mapped to “probe” memory.
Alternate Debug Vector	DebugVectorAddr	EJTAG debug, not probe, relocated, *DCR[RDVec]*==1
ROM-only entry points	`RBASE+0x0480`	EJTAG debug, when using normal ROM memory.*DCR[RDVec]*==1
	`RBASE+0x0000`	Post-reset and NMI entry point.
ROM entry points (when *Status[BEV]*==1)	`RBASE+0x0200`	Simple TLB Refill (*Status[EXL]*==0).
	`RBASE+0x0300`	Cache Error. Note that regardless of any relocation of RBASE (see above) the cache error entry point is always forced into kseg1.
	`RBASE+0x0400`	Interrupt special (*Cause[IV]*==1).
	`RBASE+0x0380`	All others
“RAM” entry points (*Status[BEV]*==0)	`BASE+0x100`	Cache error - in RAM. but always through uncached kseg1 window.
	`BASE+0x000`	Simple TLB Refill (*Status[EXL]*==0).
	`BASE+0x200`	Interrupt special (*Cause[IV]*==1).
	`BASE+0x200+...`	multiple interrupt entry points - seven more in “VI” mode, 63 in “EIC” mode; see Section 7.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)".
	`BASE+0x180`	All others

7.4 Shadow registers

In hardware terms, shadow registers are deceptively simple: just add one or more extra copies of the register file. If you can automatically change register set on an exception, the exception handler will run with its own context, and without the overhead of saving and restoring the register values belonging to the interrupted program. On to the details...

MIPS shadow registers come as one or more extra complete set of 32 general purpose registers. The CPU only changes register sets on an exception or when returning from an exception with **`eret`**.

In the 1004K CPU (and possibly other CPUs conforming to [MIPSMT]) there are no dedicated shadow registers, but you can configure the CPU to make the registers of one or more TCs available as shadow sets, as described in Section 7.4.1.

Selecting shadow sets - SRSCtl

The shadow set selectors are in the *SRSCtl* register, shown in Figure 7.3.

Figure 7.3 Fields in the SRSCtl Register

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0	HSS	0	EICSS	0	ESS	0	PSS	0	CSS
					0		0		

SRSCtl[HSS]: the highest-numbered register set available on this VPE/CPU (i.e. the number of available register sets minus one.) If it reads zero, your CPU has just one set of GPR registers and no shadow-set facility.

On single-threaded CPUs this field is fixed. However, on the 1004K CPU this field can change when configuration software - that is, when *VPEConf0[VPC]* is set - changes the way the shadow sets are shared. See Section 7.4.1 below for how multithreading TCs can be used as shadow sets.

SRSCtl[EICSS]: In EIC mode, the external interrupt controller proposes a shadow register set number with each requested interrupt (nonzero IPL). When the CPU takes an interrupt, the externally-supplied set number determines the next set and is made visible here in *SRSCtl[EICSS]* until the next interrupt.

The CPU is in EIC mode if *Config3[VEIC]* (indicating the hardware is EIC-compliant), and software has set *Cause[IV]* to enable vectored interrupts. There's more about EIC mode in Section 7.2.3 "External Interrupt Controller (EIC) mode".

If the CPU is not in EIC mode, this field reads zero.

In VI mode (no external interrupt controller, *Config3[VInt]* reads 1 and *Cause[IV]* has been set 1) the CPU sees only eight possible interrupt numbers; the *SRSMap* register contains eight 4-bit fields defining the register set to use for each of the eight interrupt levels.

If you are remaining with "classic" interrupt mode (*Cause[IV]* is zero), it's still possible to use one shadow set for all exception handlers — including interrupt handlers — by setting *SRSCtl[ESS]* non-zero.

SRSCtl[ESS]: this writable field is the software-selected register set to be used for "all other" exceptions; that's other than an interrupt in VI or EIC mode (both have their own special ways of selecting a register set).

Unpredictable things will happen if you set *ESS* to a non-existent register set number (ie, if you set it higher than the value in *SRSCtl[HSS]*.

SRSCtl[CSS,PSS]: *CSS* is the register set currently in use, and is a read-only field. It's set on any exception, replaced by the value in *SRSCtl[PSS]* on an **eret**.

PSS is the "previous" register set, which will be used following the next **eret**. It's writable, allowing the OS to dispatch code in a new register set; load this value and then execute an **eret**. If you write a larger number than the total number of implemented register sets the result is unpredictable.

You can get at the values of registers in the previous set using **rdpgpr** and **wrpgpr**.

Just a note: *SRSCtl[PSS]* and *SRSCtl[CSS]* are not updated by *all* exceptions, but only those which write a new return address to *EPC* (or equivalently, those occasions where the exception level bit *Status[EXL]* goes from zero to one). Exceptions where *EPC* is *not* written include:

- Exceptions occurring with *Status[EXL]* already set;
- Cache error exceptions, where the return address is loaded into *ErrorEPC*;
- EJTAG debug exceptions, where the return address is loaded into *DEPC*.

How new shadow sets get selected on an interrupt

In EIC mode, the external interrupt controller proposes a shadow register set number with each requested interrupt (nonzero IPL). When the CPU takes an interrupt, the externally-supplied set number determines the next set and is made visible in *SRSCtl[EICSS]* until the next interrupt.

In VI mode (no external interrupt controller) the CPU sees only eight possible interrupt numbers; the *SRSMap* register contains eight 4-bit fields, defining the register set to use for each of the eight interrupt levels, as shown in Figure 7.4.

Figure 7.4 Fields in the SRSMap Register

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0
0	0	0	0	0	0	0	0

In *SRSMap*, each of the *SSV7-0* fields has the shadow set number to be used when handling the interrupt for the corresponding *Cause[IP7-0]* bit. A zero shadow set number means not to use a shadow set. A number than the highest valid set (as found in *SRSCtl[HSS]*) has unpredictable results: don't do that.

If you are remaining with "classic" interrupt mode, it's still possible to use one shadow set for all exception handlers - including interrupt handlers - by setting *SRSCtl[ESS]* non-zero.

In "EIC" interrupt mode, this register has no effect and the shadow set number to be used is determined by an input bus from the interrupt controller.

Software support for shadow registers

Shadow registers work "as if by magic" for short interrupt routines which run entirely in exception mode (that is, with *Status[EXL]* set). The shadow registers are not just efficient because there's no need to save user registers; the shadow registers can also be used to hold contextual information for one or more interrupt routines which uses a particular shadow set. For more ambitious interrupt nesting schemes, software must save and stack copies of *SRSCtl[PSS]* alongside its copies of *EPC*; and it's entirely up to the software to determine when an interrupt handler can just go ahead and use a register set, and when it needs to save values on entry and restore them on exit. That's at least as difficult as it sounds: shadow sets are probably best used purely for very low-level, high-speed handlers.

7.4.1 Recycling multi-threading CPU's TCs as shadow sets

This recycling is controlled by some TC control bits and the *SRSConf0-4* registers.

Figure 7.5 Fields in the SRSConf0 register

31	30	29 20	19 10	9 0
M	0	SRS3	SRS2	SRS1

In *SRSConf0*:

M: is a "continuation" indication. Since there is no *SRSConf1* in the 1004K CPU, it will read zero.

In general there need be no more of these registers than are required to map your CPU's maximum complement of shadow register sets.

SRS1-3: are each set to the GPR set to be used for the putative shadow set number (1-3).

Shadow set 0 refers (in a MIPS MT CPU) to the register set normally associated with the current TC.

A value of all-ones in any of the (10-bit) *SRS1-3* fields (decimal 1023) indicates that this shadow set number is not usable - it won't select a set of registers.

The fact that there are no more “SRSConf” registers means that shadow set numbers above 4 are never usable for the 1004K CPU.

These fields may be writable (waiting to receive the number of a TC you sacrifice to provide a shadow set) or hard-wired (representing dedicated shadow register sets, whose “GPR number” will be larger than the maximum TC# of the machine.)

From reset, the writable fields take the value 1022. You just write the number of the TC you’re sacrificing. Unless the donor TC is already bound to the same VPE as owns this *SRSConf* register, nothing happens. You should also make sure the donor TC is halted, inactive and not usable by fork.

It’s possible to reverse this process and seize back a TC, so long as the shadow set concerned is no longer in use.

Note that *SRSConf0* is replicated per-VPE.

7.5 Saving Power

There are basically just a couple of facilities:

- The **`wait`** instruction: this puts the thread running to sleep. When this happens when all other threads are sleeping, halted or suspended, the CPU goes into a low-power mode with many clocks stopped, from which it will only emerge when it senses an interrupt. The interrupt will be delivered to any sleeping thread, but *all* sleeping threads will wake and return from their **`wait`**. That will usually be OK; it’s normal practice to loop over **`wait`**.

  In some CPUs — distinguished by having *Config7[WII]* set to 1 — a **`wait`** condition will be terminated by an active interrupt signal, even if that signal is prevented from causing an interrupt by *Status[IE]* being clear or *TCStatus[IXMT]* being set. It’s not immediately obvious why that behavior is useful, but it avoids a tricky race condition for an OS which uses a **`wait`** instruction in its idle loop. For programming details consult Section C.4.1 “Status register”, Section 2.9.4 “TCStatus”, and Section C.4.5 “The Config7 register”.

- The *Status[RP]* bit: this doesn’t do anything inside the CPU, but its state is made available at the CPU interface as *SI_RP*. Logic outside the CPU is encouraged to use this to control any logic which trades off power for speed - most often, that will be slowing the master clock input to the CPU.

- Via the Cluster Power Controller, it is possible to gate off the clocks or even the power going to an idle CPU. This functionality is described in [CPS_USM]

7.6 The HWREna register - Control user rdhwr access

HWREna allows the OS to control which (if any) *hardware registers* are readable in user mode using **`rdhwr`**: see also Section 6.1 “User-mode accessible “Hardware registers””.

The low four bits (3-0) relate to the four registers required by the MIPS32 standard. The two high bits (31-30) are available for implementation-dependent use.

The whole register is cleared to zero on reset, so that no hardware register is accessible without positive OS clearance.

Figure 7.6 Fields in the HWREna Register

31 30	29	28 4	3	2	1	0
Impl	UL	0	CCRes	CC	SYNCI_Step	CPUNum
0	0		0	0	0	0

HWREna[Impl]: Read 0. If there were any implementation-dependent hardware registers, you could control access to them here. Currently, no 1004K family CPU has any such extra registers.

HWREna[UL]: Set this bit 1 to permit user programs to obtain the value of the *UserLocal* CP0 register through **`rdhwr $29`**.

HWREna[CCRes]: Set this bit 1 so a user-mode **`rdhwr 3`** can determine whether *Count* runs at the full clock rate or some divisor.

HWREna[CC]: Set this bit 1 so a user-mode **`rdhwr 2`** can read out the value of the *Count* register.

HWREna[SYNCI_Step]: Set this bit 1 so a user-mode **`rdhwr 1`** can read out the cache line size (actually, the smaller of the L1 I-cache line size and D-cache line size). That line size determines the step between successive uses of the **`synci`** instruction, which does the cache manipulation necessary to ensure that the CPU can correctly execute instructions which you just wrote.

HWREna[CPUNum]: Set this bit 1 so a user-mode **`rdhwr 0`** reads out the CPU ID number, as found in *EBase[CPUNum]*.

Chapter 8

Floating point unit

The 1004Kf™ member of the 1004K family has a hardware floating point unit (FPU). This:

- *Is a 64-bit FPU*: with instructions working on both 64-bit and 32-bit floating point numbers, whose formats are compatible with the “double precision” and “single precision” recommendations of [IEEE754].

- *Is compatible with the MIPS64 Architecture*: implements the floating point instruction set defined in [MIPS64V2]; because the 1004K family integer CPU is a 32-bit processor, a couple of additional instructions **`mfhc1`** and **`mthc1`** are available to help pack and unpack 64-bit values when copying data between integer and FP registers - see Section D.3 “FPU changes in Release 2 of the MIPS32® Architecture” or for full details [MIPS32].

- *Usually runs at half the integer CPU’s clock rate*: the design is tested to work with the FPU running at the *CPU* speed, but in likely processes the FPU will then limit the achievable frequency of the whole *CPU*. You can query the *Config7[FPR]* field to check which option is used on your CPU.

- *Can run without an exception handler*: the FPU offers a range of options to handle very large and very small numbers in hardware. With the 1004K CPU full IEEE754 compliance *does* require that some operand/operation combinations be trapped and emulated, but high performance and good accuracy are available with settings which get the hardware to do everything - see Section 8.4.2, "FPU “unimplemented” exceptions (and how to avoid them)".

- *Omits “paired single” and MIPS-3D extensions:* those are primarily aimed at 3D graphics, and are described as optional in [MIPS64V2].

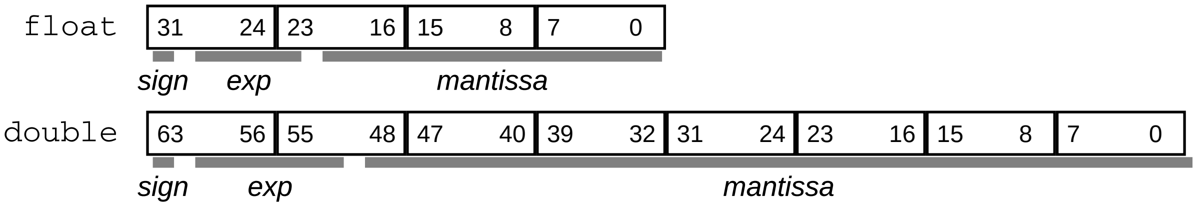
- *Uses an autonomous 7-stage pipeline*: all data transfers are interlocked, so the programmer is never aware of the pipeline. Compiler writers and daemon subroutine tuners do need to know: there’s timing information in Section 8.5, "FPU pipeline and instruction timing".

- *Has limited dual issue*: the FPU has two parallel pipelines. One handles all arithmetic operations, the other deals with loads, stores and data transfers to/from integer registers.

8.1 Data representation

If you’d like to read up on floating point in general you might like to read [SEEMIPSRUN]:. But it’s probably useful to remind you (in Figure 8.1) what 32-bit and 64-bit floating point numbers on MIPS architecture CPUs look like.

Figure 8.1 How floating point numbers are stored in a register



Just to remind you:

- *sign*: FP numbers are positive numbers with a separate sign bit; "1" denotes a negative number.
- *mantissa*: represents a binary number. But this is a floating point number, so the units depend on:
- *exp*: the exponent.

When 32-bit data is held in a 64-bit register, the high 32 bits are don't care.

The MIPS Architecture's 32-bit and 64-bit floating point formats are compatible with the definitions of "single precision" and "double precision" in [IEEE754].

FP registers can also hold simple 2s-complement signed integers too, just like the same number held in the integer registers. That happens whenever you load integer data, or convert to an integer data type.

Floating point data in memory is endianness-dependent, in just the same way as integer data is; the higher bit-numbered bytes shown in Figure 8.1 will be at the lowest memory location when the CPU is configured big-endian, and the highest memory location when the CPU is little-endian.

8.2 Basic instruction set

Whenever it makes sense to do so, FP instructions exist in a version for each data type. In assembler that's denoted by a suffix of:

.s	single-precision
.d	double-precision
.w	32-bit integer ("word")
.l	64-bit integer

There's a good readable summary of the floating point instruction set in [SEEMIPSRUN]:, and you can find the fine technical details in [MIPS64V2].

As a one-minute guide: the FPU provides basic arithmetic (add, multiply, subtract, divide and square root). It's all register-to-register (like the integer unit). It's written "destination first" like integer instructions; sometimes that's unexpected in that **`cvt.d.s`** is a "convert from single to double". It has a set of multiply/add instructions which work on *four* registers: **`madd a,b,c,d`** does

a = c*d + b

as a single operation. There are a rich set of conversion operations. A bewildering variety of compare instructions record their results in any one of eight condition flags, and there are branch and conditional-move instructions which test those flags.

You won't find any higher-level functions: no exponential, log, sine or cosine. This is a RISC instruction set, you're expected to get library functions for those things.

8.3 Floating point loads and stores

FP data does not normally pass through the integer registers; the FPU has its own load and store instructions. The FPU is conceptually a replaceable tenant of coprocessor 1: while arithmetic FP operations get recognizable names like **`add.d`**, the load/store instructions will be found under names like **`ldc1`** in [MIPS64V2] and other formal documentation. In assembler code, you'll more often use mnemonics like **`l.d`** which you'll find will work just fine.

Because FP-intensive programs are often dealing with one- or two-dimensional arrays of values, the FPU gets special load/store instructions where the address is formed by adding two registers; they're called **`ldxc1`** etc. In assembler you just use the **`l.d`** mnemonic with an appropriate address syntax, and all will be well.

8.4 Setting up the FPU and the FPU control registers

There's a fair amount of state which you set up to change the way the FPU works; this is controlled by fields in the FPU control registers, described here.

8.4.1 IEEE options

[IEEE754] defines five classes of exceptional result. For each class the programmer can select whether to get an IEEE-defined "exceptional result" or to be interrupted. Exceptional results are sometimes just normal numbers but where precision has been lost, but also can be an *infinity* or *NaN* ("not-a-number") value.

Control over the interrupt-or-not options is done through the *FCSR[Enable]* field (or more cleanly through *FENR*, the same control bits more conveniently presented); see Table 8.1 below.

It's overwhelmingly popular to keep *FENR* zero and thus never generate an IEEE exception; see Section 8.5, "FPU pipeline and instruction timing" for why this is a particularly good idea if you want the best performance.

8.4.2 FPU "unimplemented" exceptions (and how to avoid them)

It's a long-standing feature of the MIPS Architecture that FPU hardware need not support every corner-case of the IEEE standard. But to ensure proper IEEE compatibility to the software system, an FPU which can't manage to generate the correct value in every case must detect a combination of operation and operands it can't do right. It then takes an *unimplemented* exception, which the OS should catch and arrange to software-emulate the offending instruction.

The 1004K CPU's FPU will handle everything IEEE can throw at it, except for tiny numbers: it can't use or produce non-zero values which are too small for the standard ("normalized") representation[27].

[27] IEEE754 defines an alternative "denormalized" representation for these numbers.

Here you get a choice: you can either configure the CPU to depart from IEEE perfection (see the description of the *FCSR[FS,FO,FN]* bits in the notes to Section 8.1, "FPU (co-processor 1) control registers"), or provide a software emulator and resign yourself to a small number of "unimplemented" exceptions.

8.4.3 FPU control register maps

There are five FP control registers:

Table 8.1 FPU (co-processor 1) control registers

Conventional Name	*CP1 ctrl reg num*	*Description*
`FCSR`	31	Extensive control register - the only FPU control register on historical MIPS CPUs. Contains *all* the control bits. But in practice some of them are more conveniently accessed through *FCCR*, *FEXR* and *FENR* below.
`FIR`	0	FP implementation register: read-only information about the capability of this FPU.

`FCCR`
`FEXR`
`FENR`
	25
	26
	28
		Convenient partial views of *FCSR* are better structured, and allow you to update fields without interfering with the operation of independent bits.
		FCCR has FP condition codes, *FEXR* contains IEEE exceptional-condition information (cause and flag bits) you read, and *FENR* is IEEE exceptional-condition enables you write.

The FP implementation (FIR) register

Figure 8.2 shows the fields in *FIR* and the read-only values they always have for 1004K family FPUs:

Figure 8.2 Fields in the FIR register

31 25	24	23	22	21	20	19	18	17	16	15 8	7 0
0	FC	0	F64	L	W	3D	PS	D	S	Processor ID	Revision
	1		1	1	1	0	0	1	1	0x97	whatever

The fields have the following meanings:

- *FC*: "full convert range": the hardware will complete *any* conversion operation without running out of bits and causing an "unimplemented" exception.
- *F64/L/W/D/S*: this is a 64-bit floating point unit and implements 64-bit integer ("L"), 32-bit integer ("W"), 64-bit FP double ("D") and 32-bit FP single ("S") operations.
- *3D*: does not implement the MIPS-3D ASE.
- *PS*: does not implement the paired-single instructions described in [MIPS64V2]
- *Processor ID/Revision*: major and minor revisions of the FPU - as is usual with revisions it's very useful to print these out from a verbose sign-on message, and rarely a good idea to have software behave differently according to the values.

The FP control/status registers (FCSR, FCCR, FEXR, FENR)

Figure 8.3 shows all these registers and their bits

<table>
<tr><th></th><th>31 25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20 18</th><th>17</th><th>16 12</th><th>11 8</th><th>7</th><th>6 3</th><th>2</th><th>1 0</th></tr>
<tr><td>FCSR</td><td>FCC7-1</td><td>FS</td><td>FCC0</td><td>FO</td><td>FN</td><td>0</td><td>E</td><td>Cause</td><td colspan="2">Enables</td><td colspan="2">Flags</td><td>RM</td></tr>
<tr><td>FCCR</td><td colspan="9">0</td><td colspan="4">FCC7-0</td></tr>
<tr><td>FEXR</td><td colspan="6">0</td><td>E</td><td>Cause</td><td colspan="2">0</td><td colspan="2">Flags</td><td>0</td></tr>
<tr><td>FENR</td><td colspan="8">0</td><td colspan="2">Enables</td><td>0</td><td>FS</td><td>RM</td></tr>
</table>

Figure 8.3 Floating point control/status register and alternate views

Where:

FCC7-0: the floating point condition codes: set by compare instructions, tested by appropriate branch and conditional move instructions.

FS/FO/FN: options to avoid "unimplemented" exceptions when handling tiny ("denormalized") numbers[28]. They do so at the cost of IEEE compatibility, by replacing the very small number with either zero or with the nearest nonzero quantity with a normalized representation.

The *FO* ("flush override") bit causes all tiny operand and result values to be replaced.

The *FS* ("flush to zero") bit causes all tiny operand and result values to be replaced, but additionally does the same substitution for any tiny intermediate value in a multiply-add instruction. This is provided both for legacy reasons, and in case you don't like the idea that the result of a multiply/add can change according to whether you use the fused instruction or a separate multiply and add.

The *FN* bit ("flush to nearest") bit causes all result values to be replaced with somewhat better accuracy than you usually get with *FS*: the result is either zero or a smallest-normalized-number, whichever is closer. Without FN set you can only replace your tiny number with a nonzero result if the "RP" or "RM" rounding modes (round towards more positive, round towards more negative) are in effect.

For full IEEE-compatibility you must set *FCSR[FS,FO,FN]* `== [0,0,0]`.

To get the best performance compatible with a guarantee of no "unimplemented" exceptions, set *FCSR[FS,FO,FN]* `== [1,1,1]`.

Just occasionally for legacy applications developed with older MIPS CPUs which did not have the *FO* and *FN* options, you might set *FCSR[FS,FO,FN]* `== [1,0,0]`.

E: (often shown in documents as part of the *Cause* array) is a status bit indicating that the last FP instruction caused an "unimplemented" exception, as discussed in Section 8.4.2, "FPU "unimplemented" exceptions (and how to avoid them)".

[28] See [SEEMIPSRUN]: for an explanation of "normalized" and "denormalized".

Cause/Enables/Flags: each of these fields is broken up into five bits, each representing an IEEE-recognized class of exceptional results[29] which can be individually treated either by interrupting the computation, or substituting an IEEE-defined exceptional value. So each field contains:

bit number	4	3	2	1	0
field	V	Z	O	U	I

The bits are *V* for invalid operation (e.g. square root of -1), *Z* for divide-by-zero, *O* for overflow (a number too large to represent), *U* for underflow (a number too small to represent) and *I* for inexact - even `1/3` is inexact in binary.

Then the:

- *Enables* field is "write 1 to take a MIPS exception if this condition occurs" - rarely done. With the IEEE exception-catcher disabled, the hardware/emulator together will provide a suitable exceptional result.
- *Cause* field records what if any conditions occurred in the last-executed FP instruction. Because that's often too transient, the
- *Flags* field remembers all and any conditions which happened since it was last written to zero by software.

RM: is the rounding mode, as required by IEEE:

RM	*Meaning*
0	Round to nearest - *RN* If the result is exactly half-way between the nearest values, pick the one whose mantissa bit0 is zero.
1	Round toward zero - *RZ*
2	Round towards plus infinity - *RP* “Round up” (but unambiguous about what you do about negative numbers).
3	Round towards minus infinity - *RM*

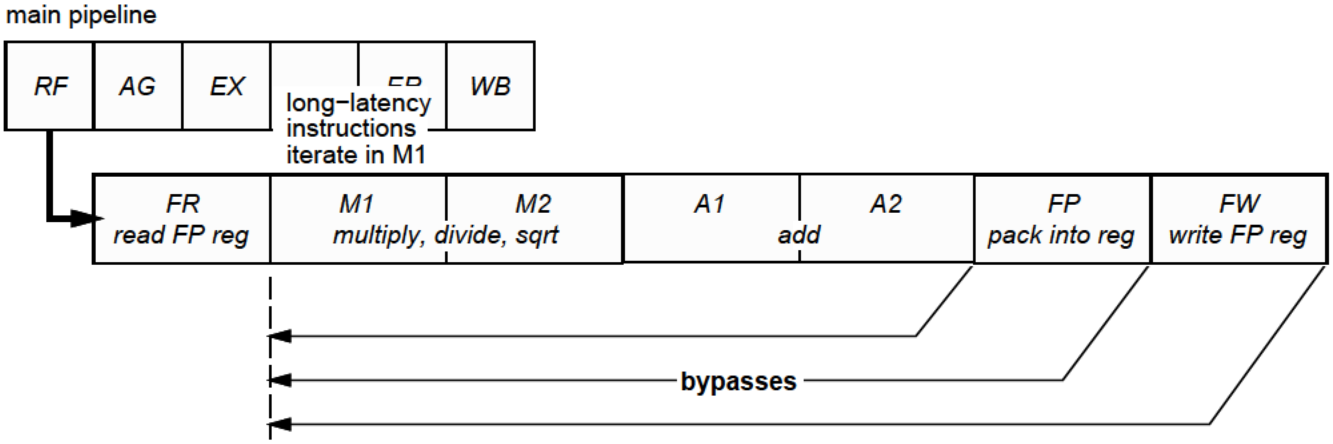
8.5 FPU pipeline and instruction timing

This is not so simple. The floating point unit (FPU) has its own pipeline. More often than not, the FPU uses a slower clock rate than the integer CPU - a full-speed FPU is a build option, but in that case the FPU will usually limit the clock rate which your design can reach. For 1004K family CPUs, the FPU will commonly be built with a half rate clock. You can find how your CPU is set up by looking at the *Config7[FPR1-0]* bits, defined in the notes to Figure C-3.

Nonetheless, this is a powerful 64-bit floating point unit which can deliver very good performance. The FPU pipeline is shown in Figure 8.4.

[29] Sorry about the ugly wording. The IEEE standard talks of “exceptions” which makes more sense but gets mixed up with MIPS “exceptions”, and they’re not the same thing.

Figure 8.4 Overview of the FPU pipeline



FPU instructions are fetched and despatched through the integer CPU. All instructions (including FP instructions) proceed through the *CPU* to graduation..

The FPU is a multiply-add pipeline, and all register-to-register instructions go through six stages:

FR: obtains FP register values and converts them into an expanded internal format.

When the FPU runs at a slower speed than the CPU, instructions issued from the integer CPU may have to wait for the next FPU clock cycle to start.

M1, M2: multiply operation as required. Some long-latency operations loop in the M1 stage until complete, holding up any subsequent FP instruction which would otherwise enter M1. Instructions issued earlier (and thus further down the pipeline) continue to run, leaving bubbles in the FP pipeline stages M2 through FW.

A1, A2: add-unit operation as required.

FP: convert result back to standard stored form and round.

FW: write back to FP register.

8.5.1 FPU register dependency delays

Any FPU instruction must go through pipeline stages from M1 through A2 before it produces a result, which can then (as shown by the "bypass" lines in the pipeline diagram) be used by a dependent instruction reaching the M1 stage. If you want to keep the FPU pipeline full, that means it's enough to have three non-dependent instructions between the consumer and producer of an FP value. However, there's no guarantee that all the FP pipeline slots will be filled, and then three intervening instructions will be excessive. Good compilers should try to schedule FP instructions, but not at unreasonable cost.

8.5.2 Delays caused by long-latency instructions looping in the M1 stage

Instructions which take only one clock in M1 go through the pipeline smoothly and can be completed one per FPU clock period. Instructions which take longer in M1 always prevent the next instruction from starting in the next clock, regardless of any data dependency. Those long-latency instructions - double-precision multiplies and all division and square root operations - are listed in Table 8.2. An instruction which runs for 2 cycles in M1 holds up the FPU pipeline for one clock and so on - and of course the cycle counts are for FPU cycles.

Table 8.2 Long-latency FP instructions

Operand	*Instruction type*	*Instructions*	*Cycles in M1*
Double-precision (64-bit)	Any multiplication	`mul.d`,`madd.d`, `msub.d`,`nmadd.d`, `nmsub.d`	2
Single-precision (32-bit)	Reciprocal	`recip.s`	10
	divide, square-root	`div.s`,`sqrt.s`	14
	reciprocal square root	`rsqrt.s`	14
Double-precision (64-bit)	Reciprocal	`recip.d`	21
	divide, square-root	`div.d`,`sqrt.d`	29
	reciprocal square root	`rsqrt.d`	31

8.5.3 Delays on FP load and store instructions

FP store instructions stall in the main pipeline EX stage until the register data arrives from the FPU. Provided that the store instruction doesn’t get behind slow FP instructions, FP stores run no more than one every two instructions and should not produce further delays.

FP load instructions are subject to the usual FPU timing. So long as the load hits in the cache, you should see no more than the usual FP producer-consumer delay from load to use.

8.5.4 Delays when main pipeline waits for FPU to decide not to take an exception

The MIPS architecture requires FP exceptions to be “precise”, which (in particular) means that no instruction after the FP instruction causing the exception may do anything software-visible. That means that an FP instruction in the main pipeline may not be committed, nor leave the main pipeline, until the FPU can either report the exception, or confirm that the instruction will not cause an exception.

Floating point instructions cause exceptions not only because a user program has requested the system to trap IEEE exceptional conditions (which is unusual) but also because the hardware is not capable of generating or accepting very small (“denormalized”) numbers in accordance with the IEEE standards. The latter (“unimplemented”) exception is used to call up a software emulator to patch up some rare cases. But the main pipeline must be stalled until the FP hardware can rule out an exception, and that leads to a delay on every non-trivial FP operation. With a half-rate FPU, this stall will most likely be 6-7 clocks.

Software which can tolerate some deviation from IEEE precision can avoid these delays by opting to replace all denormalized inputs and results by zero - controlled by the *FCSR[FS,FO,FN]* register bits described in Section 8.1, "FPU (co-processor 1) control registers" and its notes. If you have also disabled all IEEE traps, you get no possibility of FP exceptions and no extra main pipeline delay.

8.5.5 Delays when main pipeline waits for FPU to accept an instruction

An FPU running slower than the *CPU* can only accept instructions on the appropriate clock edges: back to back FP instructions will cause delays. But if some of your FP instructions are the long-latency ones described above, the FP pipeline has room for just one more instruction before it backs up. Once it does back up, your whole CPU will stall until the long-latency instruction completes.

8.5.6 Delays on mfc1/mtc1 instructions

Any FP instruction with GP register operands gets sent the GP values when it is launched, so **`mtc1`** instructions have standard FP instruction timing. An **`mfc1`** instructions needs to write data into the GP register file. In general it will not complete quickly enough to use its main-pipeline register file write slot, so the value returning to the integer unit must wait until the integer unit is not using the GP register write port. The instruction which uses the value obtained by the **`mfc1`** may stall until the data is available, but that usually won't be very long.

8.5.7 Delays caused by dependency on FPU status register fields

The conditional branch instructions **`bc1f`**/**`bc1t`** and the conditional moves **`movf`**/**`movt`** execute in the main pipeline, but test a FP condition bit generated by the various FPU compare instructions.

8.5.8 Slower operation in MIPS I™ compatibility mode

Historic 32-bit MIPS CPUs had only 16 "even-numbered" floating point registers usable for arithmetic, with odd-numbered registers working together with them to let you load, store and transfer double-precision (64-bit) values. Software written for those old CPUs is incompatible with the full modern FPU, so there's a compatibility bit provided in *Status[FR]* - set zero to use MIPS I compatible code. This comes at the cost of slower repeat rates for FP instructions, because in compatibility mode not all the bypasses shown in the pipeline diagram above are active.

Chapter 9

The MIPS32® DSP ASE

The MIPS DSP ASE is provided to accelerate a large range of DSP algorithms. You can get most programming information from this chapter. There's more detail in the formal DSP ASE specification [MIPSDSP], but expect to read through lots of material aimed at hardware implementors. You may also find [DSPWP] useful for tips and examples of converting DSP algorithms for the DSP ASE.

Different target applications generally need different data size and precision:

- *32-bit data*: audio (non-hand-held) decoding/encoding - a wide range of "hi-fi" standards for consumer audio or television sound.

  Raw audio data (as found on CD) is 16-bit; but if you do your processing in 16 bits you lose precision beyond what is acceptable for hi-fi.

- *16-bit data*: digital voice for telephony. International telephony code/decode standards include G.723.1 (8Ksample/s, 5-6Kbit/s data rate, 37ms delay), G.729 (8Kbit/s, 15ms delay) and G.726 (16-40Kbit/s, computationally simpler and higher quality, good for carrying analogue modem tones). Application-specific filters are used for echo cancellation, noise cancellation, and channel equalization.

  Also used for soft modems and much general "DSP" work (filters, correlation, convolution); lo-fi devices use 16 bits for audio.

- *8-bit data*: processing of printer images, JPEG (still) images and video data.

9.1 Features provided by the MIPS® DSP ASE

Those target applications can benefit from unconventional architecture features because they rely on:

- *Fixed-point fractional data types*: It is not yet economical (in terms of either chip size or power budget) to use floating point calculations in these contexts. DSP applications use fixed-point fractions. Such a fraction is just a signed integer, but understood to represent that integer divided by some power of two. A 32-bit fractional format where the implicit divisor is 2^{16} (65536) would be referred to as a Q15.16 format; that's because there are 16 bits devoted to fractional precision and 15 bits to the whole number range (the highest bit does duty as a sign bit and isn't counted).

  With this notation Q31.0 is a conventional signed integer, and Q0.31 is a fraction representing numbers between -1 and 1 (well, nearly 1). It turns out that Q0.31 is the most popular 32-bit format for DSP applications, since it won't overflow when multiplied (except in the corner case where -1×-1 leads to the just-too-large value 1). Q0.31 is often abbreviated to Q31.

  The DSP ASE provides support for Q31 and Q15 (signed 16-bit) fractions.

- *Saturating arithmetic*: It's not practicable to build in overflow checks to DSP algorithms - they need to be too fast. Clever algorithms may be built to be overflow-proof; but not all can be. Often the least worst thing to do when a calculation overflows is to make the result the most positive or most negative representable value. Arithmetic which does that is called *saturating* - and quite a lot of operations in the DSP ASE saturate (in many cases there are saturating and non-saturating versions of what is otherwise the same instruction).

- *Multiplying fractions*: if you multiply two Q31 fractions by re-using a full-precision integer multiplier, then you’ll get a 64-bit result which consists of a Q62 result with (in the very highest bit) a second copy of the sign bit. This is a bit peculiar, so it’s more useful if you always do a left-shift-by-1 on this value, producing a Q63 format (a more natural way to use 64 bits). Q15 multiplies which generate a Q31 value have to do the shift-left too. That’s what all the **`mulq`**... instructions do.

- *Rounding*: some fractional operations implicitly discard less significant bits. But you get a better approximation if you bump the truncated result by one when the discarded bits represent more than a half of the value of a 1 in the new LS position. That’s what we mean by *rounding* in this chapter.

- *Multiply-accumulate sequences with choice of four accumulators*: (with fixed-point types, sometimes saturating).

  The 1004K already has quite a slick integer multiply-accumulate operation, but it’s not so efficient when used for fractional and saturating operations.

  The sequences are made more usable by having four 64-bit result/accumulator registers - (the old MIPS multiply divide unit has just one, accessible as the *hi*/*lo* registers). The new *ac0* is the old *hi*/*lo*, for backward compatibility.

- *Benefit from “SIMD” operations.*: Many DSP calculations are a good match for “Single Instruction Multiple Data” or *vector* operations, where the same arithmetic operation is applied in parallel to several sets of operands.

  In the MIPS DSP ASE, some operations are SIMD type - two 16-bit operations or four 8-bit operations are carried out in parallel on operands packed into a single 32-bit general-purpose register. Instructions operating on vectors can be recognized because the name includes **`.ph`** (paired-half, usually signed, often fractional) or **`.qb`** (quad-byte, always unsigned, only occasionally fractional).

The DSP ASE hardware involves an extensive re-work of the normal integer multiply/divide unit. As mentioned above it has four 64-bit accumulators (not just one) and a new control register, described immediately below.

9.2 The DSP ASE control register

This is a part of the user-mode programming model for the DSP ASE, and is a 32-bit value read and written with the **`rddsp`**/**`wrdsp`** instructions. It holds state information for some DSP sequences.

Figure 9.1 Fields in the DSPControl Register

31 28	27 24	23 16	15	14	13	12 7	6	5 0
0	ccond	ouflag	0	EFI	c	scount	0	pos

In Figure 9.1:

ccond: condition bits set by compare instructions (there have to be four to report on compares between vector types). "Compare" operations on scalars or vectors of length two only touch the lower-numbered bits. *DSPControl* bits 31:28 are used for more *ccond* bits in 64-bit machines.

ouflag: one of these bits may be set when a result overflows (whether or not the result is saturated depends on the instruction - the flag is set in either case). The "ou" stands for "overflow/underflow" - "underflow" is used here for a value which is negative but with excessive absolute value.

Any overflowed/underflowed result produced by any DSP ASE instruction sets a *ouflag* bit, *except* for **`addsc`**/**`addwc`** and **`shilo`**/**`shilov`**.

The 6 bits are set according to the destination of the operation which overflowed, and the kind of operation it was:

Bit No	*Overflowed destination/instruction*
16-19	Destination register is a multiply unit accumulator: separate bits are respectively for accumulators 0-3.
20	Add/subtract.
21	Multiplication of some kind.
22	Shift left or conversion to smaller type
23	Accumulator shift-then-extract

EFI: set by any of the accumulator-to-register bitfield extract instructions **`extp`**, **`extpv`**, **`extpdp`**, or **`extpdp`**. It’s set to 1 if and only if the instruction finds there are insufficient bits to extract. That is, if *DSPControl[pos]* - which is supposed to mark the highest-numbered bit of the field we’re extracting - is less than the size value specified by the instruction.

c: Carry bit for 32-bit add/carry instructions **`addsc`** and **`addwc`**.

scount, pos: Fields for use by "variable" bitfield insert and extract instructions, such as **`insv`** (the normal MIPS32 **`ins`**/**`ext`** instructions have the field size and position hard-coded in the instruction).

scount specifies the size of the bit field to be inserted, while *pos* specifies the insert position.

Caution: in all inserts (following the lead of the standard MIPS32 insert/extract instructions) *pos* is set to the lowest bit number in the field. But in the DSP ASE extract-from-accumulator instructions (**`extp`**, **`extpv`**, **`extpdp`** and **`extpdpv`**), *pos* identifies the *highest*-numbered bit in the field.

The latter two (“dp”) instructions post-decrement *pos* (by the bitfield length *size*), to help software which is unpacking a series of bitfields from a dense data structure.

The **`mthlip`** instruction will increment the *pos* value by 32 after copying the value of *lo* to *hi*.

9.2.1 DSP accumulators

Whereas a standard MIPS32 architecture CPU has just one 64-bit multiply unit accumulator (accessible as *hi*/*lo*), the DSP ASE provides three 64-bit accumulators. Instructions accessing the extra accumulators specify a 2-bit field as 0-3 (0 selects the original accumulator).

9.3 Software detection of the DSP ASE

You can find out if your core supports the DSP ASE by testing the *Config3[DDSP]* bit (see notes to Figure 4.4).

Then you need to enable use of instructions from the MIPS DSP ASE by setting *Status[MX]* (or its alternate view *TCStatus[TMX]*) to 1.

9.4 DSP instructions

The DSP instruction set is nothing like the regular and orthogonal MIPS32 instruction set. It's a collection of special-case instructions, in many cases aimed at the known hot-spots of important algorithms.

We'll summarize the instructions under headings, but then list all of them in Section 9.2, "DSP instructions in alphabetical order", an alphabetically-ordered list which provides a terse but usually-sufficient description of what each instruction does.

9.4.1 Hints in instruction names

An instruction's name may have some suffixes which are often informative:

`q`: generally means it treats operands as fractions (which isn't important for adds and subtracts, but is important for multiplications and convert operations);

`_s`: usually means the full-precision result is saturated to the size of the destination; **`_sa`** is used for instructions which saturate intermediate results before accumulating; and **`r`**: denotes rounding (see above);

`.w, .ph, .qb`: suggest the operation is dealing with 32-bit, paired-half or quad-byte values respectively. Where there are two of these (as in **`macq_s.w.phl`**) the first one suggests the type of the result, and the second the type of the operand(s).

`v`: (in a shift instruction) suggests that the shift amount is defined in a register, rather than being encoded in a field of the instruction.

To help you get your arms around this collection of instructions we'll group them by likely usage - guided by the type of the result performed, with an eye to the application. The multiplication instructions are more tricky: most of them have multiple uses. We've sorted them by the most obvious use (likely also the most common). The classification we've chosen divides them into:

- Arithmetic - 64-bit
- Arithmetic - saturating and/or SIMD Types
- Bit-shifts - saturating and/or SIMD types
- Comparison and "conditional-move" operations on SIMD types - includes **`pick`** instructions.
- Conversions to and from SIMD types
- Multiplication - SIMD types with result in GP register
- Multiply Q15s from paired-half and accumulate
- Load with register+register address
- DSPControl register access
- Accumulator access instructions
- Dot products and building blocks for complex multiplication - includes full-word (Q31) multiply-accumulate
- Other DSP ASE instructions - everything else...

9.4.2 Arithmetic - 64-bit

addsc/**addwc** generate and use a carry bit, for efficient 64-bit add.

9.4.3 Arithmetic - saturating and/or SIMD Types

- *32-bit signed saturating arithmetic*: **addq_s.w**, **subq_s.w** and **absq_s.w**.
- *Paired-half and quad-byte SIMD arithmetic*: perform the same operation simultaneously on both 16-bit halves or all four 8-bit bytes of a 32-bit register. The "**q**" in the instruction mnemonic for the PH operations here is cosmetic: Q15 and signed 16-bit integer add/subtract operations are bit-identical - Q15 only behaves very differently when converted or multiplied.

  The paired half operations are: **addq.ph**/**addq_s.ph**, **subq.ph**/**subq_s.ph** and **absq_s.ph**.

  The quad-byte operations (all unsigned) are: **addu.qb**/**addu_s.qb**, **subu.qb**/**subu_s.qb**.
- *Sum of quad-byte vector*: **raddu.w.qb** does an unsigned sum of the four bytes found in a register, zero extends the result and delivers it as a 32-bit value.

9.4.4 Bit-shifts - saturating and/or SIMD types

All shifts can either have a shift amount encoded in the instruction, or - indicated by a trailing "**v**" in the instruction name - provided as a register operand. PH and 32-bit shifts have optional forms which saturate the result.

- *32-bit signed shifts*: include a saturating version of shift left, **shll_s.w**; and an auto-rounded shift right (just the "arithmetic", sign-propagating form): **shra_r.w**. Recall from above that rounding can be imagined as pre-adding a half to the least significant surviving bit.
- *Paired-half and quad-byte SIMD shifts*: **shll.ph**/**shllv.ph**/**shll_s.ph**/**shllv_s** are as above. For PH only there's a shift-right-arithmetic instruction ("arithmetic" means it propagates the sign bit downward) **shra.ph**, which has a variant which rounds the result **shra_r.ph**.

  The quad-byte shifts are unsigned and don't round or saturate: **shll.qb**/**shllv.qb**, **shrl.qb**/**shrlv.qb**.

9.4.5 Comparison and "conditional-move" operations on SIMD types

The "**cmp**" operations simultaneously compare and set flags for two or four values packed in a vector (with equality, less-than and less-than-or-equal tests). For PH that's **cmp.eq.ph**, **cmp.lt.ph** and **cmp.le.ph**. The result is left in the two LS bits of *DSPControl[ccond]*.

For quad-byte values **cmpu.eq.qb**, **cmpu.lt.qb** and **cmpu.le.qb** simultaneously compare and set flags for four bytes in *DSPControl[ccond]* - the flag relating to the bytes found in the low-order bits of the source register is in the lowest-numbered bit (and so on). There's an alternative set of instructions **cmpgu.eq.qb**, **cmpgu.lt.qb** and **cmpgu.le.qb** which leave the 4-bit result in a specified general-purpose register.

pick.ph uses the two LS bits of *DSPControl[ccond]* (usually the outcome of a paired-half compare instruction, see above) to determine whether corresponding halves of the result should come from the first or second source register. Among other things, this can implement a paired-half conditional move. You can reverse the order of your conditional inputs to do a move dependent on the complementary condition, too.

pick.qb does the same for QB types, this time using four bits of *DSPControl[ccond]*.

9.4.6 Conversions to and from SIMD types

Conversion operations from larger to smaller fractional types have names which start “**`precrq...`**” for “precision reduction, fractional”. Conversion operations from smaller to larger have names which start “**`prece...`**” for “precision expansion”.

- *Form vector from high/low parts of two other paired-half values*: **`packrl.ph`** makes a paired-half vector from two half vectors, swapping the position of each sub-vector. It can be used to acquire a properly formed sub-vector from a non-aligned data stream.

- *One Q15 from a paired-half to a Q31 value*: **`preceq.w.phl`**/**`preceq.w.phr`** select respectively the “left” (high bit numbered) or “right” (low bit numbered) Q15 value from a paired-half register, and load it into the result register as a Q31 (that is, it’s put in the high 16 bits and the low 15 bits are zeroed).

- *Two bytes from a quad-byte to paired-half*: **`precequ.ph.qbl`**/**`precequ.ph.qbr`** picks two bytes from either the “left” (high bit numbered) or “right” (low bit numbered) halves of a quad-byte value, and unpacks to a pair of Q15 fractions.

  `precequ.ph.qbla` does the same, except that it picks two “alternate” bytes from bits 31-24 and 15-8, while **`precequ.ph.qbra`** picks bytes from bits 23-16 and 7-0.

  Similar instructions without the **`q`** - **`preceu.ph.qbl`**, **`preceu.ph.qbr`**, **`preceu.ph.qbla`**” and **`preceu.ph.qbra`** - work on the same register fields, but treat the quantities as integers, so the 16-bit results get their low bits set.

- *2*×Q31 to a paired-half: both operands and result are assumed to be signed fractions, so **`precrq.ph.w`** just takes the high halves of the two source operands and packs them into a paired-half; **`precrq_rs.ph.w`** rounds and saturates the results to Q15.

- *2*×paired-half to quad-byte: you need two source registers to provide four paired-half values, of course. This is a fractional operation, so it’s the low bits of the 16-bit fractions which are discarded.

  `precrq.qb.ph` treats the paired-half operands as unsigned fractions, retaining just the 8 high bits of each 16-bit component.

  `precrqu_s.qb.ph` treats the paired-half operands as Q15 signed fractions and both rounds and saturates the result (in particular, a negative Q15 fraction produces a zero byte, since zero is the lowest representable quantity).

- *Replicate immediate or register value to paired-half*: in **`repl.ph`** the value to be replicated is a 10-bit signed immediate value (that’s in the range $-512 \le x \le 511$) which is sign-extended to 16 bits, whereas in **`replv.ph`** the value - assumed to be already a Q15 value - is in a register.

- *Replicate single value to quad-byte*: there’s both a register-to-register form **`replv.qb`** and an immediate form **`repl.qb`**.

9.4.7 Multiplication - SIMD types with result in GP register

When a multiply’s destination is a general-purpose register, the operation is still done in the multiply unit, and you should expect it to overwrite the *hi*/*lo* registers (otherwise known as *ac0*.)

- *8-bit*×16-bit 2-way SIMD multiplication: **`muleu_s.ph.qbl`**/**`muleu_s.ph.qbr`** picks the “left” (high bit numbered) or “right” (low bit numbered) pair of byte values from one source register and a pair of 16-bit values from the other. Two unsigned integer multiplications are done at once, the results unsigned-saturated and delivered to the two 16-bit halves of the destination.

  The asymmetric use of the source operands is not a bit like a Q15 operation. But 8×16 multiplies are heavily used in imaging and video processing (JPEG image encode/decode, for example).

- *Paired-half SIMD multiplication*: **`mulq_rs.ph`** multiplies two Q15s at once and delivers it to a paired-half value i n a general-purpose register, with rounding and saturation.

- *Multiply half-PH operands to a Q31 result*: **`muleq_s.w.phl`**/**`muleq_s.w.phr`** pick the “left”/”right” Q15 value respectively from each operand, multiply and store a Q31 value.

  “Precision-doubling” multiplications like this *can* overflow, but only in the extreme case where you multiply -1×-1, and can’t represent 1 exactly.

9.4.8 Multiply Q15s from paired-half and accumulate

`maq_s.w.phl`/**`maq_s.w.phr`** picks either the left/high or right/low Q15 value from each operand, multiplies them to Q31 and accumulates to a Q32.31 result. The multiply is saturated only when it’s -1×-1.

`maq_sa.w.phl`/**`maq_sa.w.phr`** differ in that the final result is saturated to a Q31 value held in the low half of the accumulator (required by some ITU voice encoding standards).

9.4.9 Load with register + register address

Previously available only for floating point data[30]: **`lwx`** for 32-bit loads, **`lhx`** for 16-bit loads (sign-extended) and **`lbux`** for 8-bit loads, zero-extended.

9.4.10 DSPControl register access

`wrdsp rs,mask` sets *DSPControl* fields, but only those fields which are enabled by a 1 bit in the 6-bit mask.

`rddsp` reads *DSPControl* into a GPR; but again it takes a mask field. Bitfields in the GPR corresponding to *DSPControl* fields which are not enabled will be set all-zero.

The mask bits tie up with fields like this:

Table 9.1 Mask bits for instructions accessing the DSPControl register

Mask Bit	*DSPControl field*
0	`pos`
1	`scount`
2	`c`
3	`ouflag`
4	`ccond`
5	`EFI`

[30] Well, an integer instruction is also included in the MIPS SmartMIPS™ ASE.

9.4.11 Accumulator access instructions

- *Historical instructions which now access new accumulators*: the familiar **mfhi**/**mflo**/**mthi**/**mtlo** instructions now take an optional extra accumulator-number parameter.

- *Shift and move to general register*: **extr.w**/**extr_r.w**/**extr_rs.w** gets a 32-bit field from an accumulator (starting at bit 0 up to 31) and puts the value in a general purpose register. At your option you can specify rounding and signed 32-bit saturation.

  extrv.w/**extrv_r.w**/**extrv_rs.w** do the same but specify the field's starting bit number with a register.

- *Extract bitfield from accumulator*: **extp**/**extpv** takes a bitfield (up to 32 bits) from an accumulator and moves it to a GPR. The length of the field can be an immediate value or from a register. The position of the field is determined by *DSPControl[pos]*, which holds the bit number of the most significant bit.

  extpdp/**extpdpv** do the same, but also auto-decrement *DSPControl[pos]* to the bit-number just below the field you extracted.

- *Accumulator rearrangement*: **shilo**/**shilov** has a *signed* shift value between -32 and +31, where positive numbers shift right, and negative ones shift left. The "v" version, as usual, takes the shift value from a register. The right shift is a "logical" type so the result is zero extended.

- *Fill accumulator pushing low half to high*: **mthlip** moves the low half of the accumulator to the high half, then writes the GPR value in the low half. Generally used to bring 32 more bits from a bitstream into the accumulator for parsing by the various **ext**... instructions.

9.4.12 Dot products and building blocks for complex multiplication

In 2-dimensional vector math (or in any doubled-up step of a multiply-accumulate sequence which has been optimized for 2-way SIMD) you're often interested in the *dot product* of two vectors:

```
v[0]*w[0] + v[1]*w[1]
```

In many cases you take the dot product of a series of vectors and add it up, too.

Some algorithms use complex numbers, represented by 2D vectors. Complex numbers use `i` to stand for "the square root of -1", and a vector `[a,b]` is interpreted as `a+ib` (mathematicians leave out the multiply sign and use single-letter variables, habits which would not be appreciated in C programming!) Complex multiplication just follows the rules of multiplying out sums, remembering that `i*i=-1`, so:

```
(a + ib)*(c + id) = (a*c - b*d) + i(a*d + b*c)
```

Or in vector format:

```
[a, b] * [c, d] = [a*c - b*d, a*d + b*c]
```

The first element of the result (the "real component") is like a dot product but with a subtraction, and the second (the "imaginary component") is like a dot product but with the vectors crossed.

- *Q15 dot product from paired-half, and accumulate*: **`dpaq_s.w.ph`** does a SIMD multiply of the Q15 halves of the operands, then adds the results and saturates to form a Q31 fraction, which is accumulated into a Q32.31 fraction in the accumulator.

  `dpsq_s.w.ph` does the same but subtracts the dot product from the accumulator.

  For the imaginary component of a complex multiply, first swap the Q15 numbers in one of the register operands with a **`rot`** (bit-rotate) instruction.

  For the real component of a complex Q15 multiply, you have the difference-of-products instruction **`mulsaq_s.w.ph`**, which parallel-multiplies both Q15 halves of the PH operands, then computes the difference of the two results and leaves it in an accumulator in Q32.31 format (beware: this does not accumulate the result).

- *16-bit integer dot-product from paired-half, and accumulate*: **`dpau.h.qbl`**/**`dpau.h.qbr`** picks two QB values from each source register, parallel-multiplies the corresponding pairs to integer 16-bit values, adds them together and then adds the whole lot into an accumulator. **`dpsu.h.qbl`**/**`dpsu.h.qbr`** do the same sum-of-products, but the result is then subtracted from the accumulator. In both cases, note this is integer (not fractional) arithmetic.

- *Q31 saturated multiply-accumulate*: is the nearest thing you can get to a dot-product for Q31 values. **`dpaq_sa.l.w`** does a Q31 multiplication and saturates to produce a Q63 result, which is added to the accumulator and saturated again. **`dpsq_sa.l.w`** does the same, except that the multiply result is subtracted from the accumulator (again, useful for the real component of a complex number).

9.4.13 Other DSP ASE instructions

- *Branch on DSPControl field*: **`bposge32`** branches if *DSPControl[pos]*≥`32`.

  Typically the test is for “is it time to load another 32 bits of data from the bitstream yet?”.

- *Circular buffer index update*: **`modsub`** takes an operand which packs both a maximum index value and an index step, and uses it to decrement a “buffer index” by the step value, but arranging to step from zero to the provided maximum.

- *Bitfield insert with variable size/position*: **`insv`** is a bit-insert instruction. It acts like the MIPS32 standard instruction **`ins`** except that the position and size of the inserted field are specified not as immediates inside the instruction, but are obtained from *DSPControl[pos]* (which should be set to the lowest numbered bit of the field you want) and *DSPControl[scount]* respectively.

- *Bit-order reversal*: **`bitrev`** reverses the bits in the low 16 bits of the register. The high half of the destination is zero.

  The bit-reverse operation is a computationally crucial step in buffer management for FFT algorithms, and a 16-bit operation supports up to a 32K-point FFT, which is much more than enough. A full 32-bit reversal would be expensive and slow.

9.5 Macros and typedefs for DSP instructions

It’s useful to be able to use fragments of C code to describe what some instructions do. To do that, we need to be able to refer to fractional types, saturation and vectors. Here are the definitions we’re using[31]:

```
typedef long long int64;
typedef int int32;

/* accumulator type */
typedef signed long long q32_31;

typedef signed int q31;

#define MAX31 0x7FFFFFFF
#define MIN31 -(1<<31)
#define SAT31(x) (x > MAX31 ? MAX31: x < MIN31 ? MIN31: x)

typedef signed short q15;
#define MAX15 0x7FFF
#define MIN15 -(1<<15)
#define SAT15(x) (x > MAX15 ? MAX15: x < MIN15 ? MIN15: x)

typedef unsigned char u8;
#define MAXUBYTE 255
#define SATUBYTE(x) (x > MAXUBYTE ? MAXUBYTE: x < 0 ? 0: x)

/* fields in the vector types are specified by relative bit
   position, but C definitions are in memory order, so these
   definitions need to be endianness-dependent */

#ifdef BIG_ENDIAN
typedef struct{
  q15 h1, h0;
} ph;

typedef struct{
  u8 b3, b2, b1, b0;
} qb;
#else
typedef struct{
  q15 h0, h1;
} ph;

typedef struct{
  u8 b0, b1, b2, b3;
} qb;
#endif
```

9.6 Almost Alphabetically-ordered table of DSP ASE instructions

Table 9.2 DSP instructions in alphabetical order

Instruction	*Description*
absq_s.w rd,rt	Q31/signed integer absolute value with saturation
addq.ph rd,rs,rt **addq_s.ph rd,rs,rt**	2×SIMD Q15 addition, without and with saturation of the result
addq_s.w rd,rs,rt	Q31/signed integer addition with saturation

[31] This page needs more work, and I hope it will be improved in a future version of the manual.

Table 9.2 DSP instructions in alphabetical order

Instruction	*Description*
addsc rd,rs,rt **addwc rd,rs,rt**	Add setting carry, then add with carry. The carry bit is kept in *DSPControl[c]*. So to add the 64-bit values in registers *yhi*/*ylo*, *zhi*/*zlo* to produce a 64-bit value in *xhi*/*xlo*, just do: `addsc xlo, ylo, zlo; addwc xhi, yhi, zhi`
addu.qb rd,rs,rt **addu_s.qb rd,rs,rt**	4×SIMD QBYTE addition, without and with SATUBYTE saturation.
bitrev rd,rt	Delivers the bit-reversal of the low 16 bits of the input (result has high half zero).
bposge32 offset	Branch if *DSPControl[pos]*>=32. Like most branch instruction, it has a 16-bit "PC-relative" target encoding.
cmp.eq.ph rs,rt **cmp.le.ph rs,rt** **cmp.lt.ph rs,rt**	Signed compare of both halves of two paired-half ("PH") values. Results are written into *DSPControl[ccond1-0]* for high and low halves respectively (1 for true, 0 for false). A signed compare works for both Q15 or signed 16-bit values.
cmpgu.eq.qb rd,rs,rt **cmpgu.le.qb rd,rs,rt** **cmpgu.lt.qb rd,rs,rt**	Unsigned simultaneous compare of all four bytes in quad-byte values. The four result bits are written into the four LS bits of general register *rd*.
cmpu.eq.qb rs,rt **cmpu.le.qb rs,rt** **cmpu.lt.qb rs,rt**	Unsigned simultaneous compare of all four bytes in quad-byte values. The four result bits are written into register *DSPControl[cond3-0]*.
dpaq_s.w.ph ac,rs,rt	"Dot product and accumulate", with Q31 saturation of each multiply result: `ph rs,rt; ac += SAT31(rs.h0*rt.h0 + rs.h1*rt.h1);` The accumulator is effectively used as a Q32.31 fraction.
dpaq_sa.l.w ac,rs,rt	Q31 saturated multiply-accumulate

dpau.h.qbl
 	
dpau.h.qbr
	`qb rs, rt;`
	`ac += rs.b3*rt.b3 + rs.b2*rt.b2;`
	Dot-product and accumulate of quad-byte values ("l" for left, because these are the higher bit-numbered bytes in the 32-bit register).
	Not a fractional computation, just unsigned 8-bit integers.
	Then for the lower bit-numbered bytes:
	`qb rs, rt;`
	`ac += rs.b1*rt.b1 + rs.b0*rt.b0;`

dpsq_s.w.ph ac,rs,rt	Paired-half fractional "dot product and subtract from accumulator"
	`ph rs, rt;`
	`q32_31 ac;`
	`ac -= SAT31(rs.h1*rt.h1 + rs.h0*rt.h0);`

dpsq_sa.l.w ac,rs,rt	Q31 saturated fractional-multiply, then subtract from accumulator:
	`q31 rs, rt; q32_31 ac;`
	`ac -= SAT31(rs*rt);`

dpsu.h.qbl ac,rs,rt
 	
dpsu.h.qbr ac,rs,rt
	QB format dot-product and subtract from accumulator. This is an integer (not fractional) multiplication and comes in "left" and "right" (higher/lower-bit numbered pair) versions:
	`qb rs,rt;`
	`ac -= rs.b3*rt.b3 + rs.b2*rt.b2;`
	`qb rs,rt;`
	`ac -= rs.b1*rt.b1 + rs.b0*rt.b0;`

extp rt,ac,size **extpdp rt,ac,size** **extpdpv rt,ac,rs** **extpv rt,ac,rs**	Extract bitfield from an accumulator to register. The length of the field (number of bits) can be an immediate constant or can be provided by a second source register (in the **v** variants). The field position, though, comes from *DSPControl[pos]*, which marks the highest-numbered bit of the field (note that the MIPS32 standard bitfield extract instructions specify the *lowest* bit number in the field). In the **dp** variants like **extpdp**/**extpdpv**, *DSPControl[pos]* is auto-decremented by the length of the field extracted, which is useful when unpacking the accumulator into a series of fields.

Table 9.2 DSP instructions in alphabetical order

Instruction	*Description*

extr.w rt,ac,shift
extr_r.w rt,ac,shift
extr_rs.w rt,ac,shift
extrv.w rt,ac,rs
extrv_r.w rt,ac,rs
extrv_rs.w rt,ac,rs
	Extracts a bit field from an accumulator into a general purpose register. The LS bit of the extracted field can start anywhere from bit zero to 31 of the accumulator:
	`int64 ac; unsigned int rt;`
	`rt = (ac >> shift) & 0xFFFFFFFF;`
	At option you can specify rounding (**_r** names):
	`int64 ac; unsigned int rt;`
	`rt = ((ac + 1<<(shift-1)) >> shift) & 0xFFFFFFFF;`
	and signed 32-bit saturation of the result (**_s**/**_rs** names).
	The **extrv...** variants specify the shift amount (still limited to 31 positions) with a register.

extr_s.h rt,ac,shift **extrv_s.h rt,ac,rs**	Obtain a right-shifted value from an accumulator and form a signed 16-bit saturated result.
insv rt,rs	The bitfield insert in the standard MIPS32 instruction set is **ins rt,rs,pos,size**, and the position and size must be constants (encoded as immediates in the instruction itself). This instruction permits the position and size to be calculated by the program, and then supplied as *DSPControl[pos]* and *DSPControl[scount]* respectively. In this case *DSPControl[pos]* must be set to the *lowest* numbered bit in the field to be inserted: yes, that's different from the **extp...** instructions.
lbux rd,index(base) **lhx rd,index(base)** **lwx rd, index(base)**	Load operations with register+register address formation. **lbux** is a load byte and zero extend, **lhx** loads half-word and sign-extends, and **lwx** loads a whole word. The full address must be naturally aligned for the data type.
maq_s.w.phl ac,rs,rt **maq_s.w.phr ac,rs,rt** **maq_sa.w.phl ac,rs,rt** **maq_sa.w.phr ac,rs,rt**	Non-SIMD Q15 multiply-accumulate, with operands coming from either the "left" (higher bit number) or "right" (lower bit number) half of each of the operand registers. In all versions the Q15 multiplication is saturated to a Q31 results. The "_sa" variants saturates the add result in the accumulator to a Q31, too.
mfhi rd, ac **mflo rd, ac**	Legacy instruction, which now works on new accumulators (if you provide a second nonzero argument). Copies high/low half (respectively) of accumulator to *rd*.
modsub rd,rs,rt	Circular buffer index update. *rt* packs both the decrement amount (low 8 bits) and the highest index (high 24 bits), then this instruction calculates: `rd = (rs == 0) ?   ((unsigned) rt >> 8): rs - (rt & 0xFF);`
mthi rs, ac	Legacy instruction working on new accumulators. Moves data from *rd* to the high half of an accumulator.
mthlip rs, ac	Moves the low half of the accumulator to the high half, then writes the GPR value in the low half.
mtlo rs, ac	Legacy instruction working on new accumulators. Moves data from *rd* to the low half of an accumulator.
muleq_s.w.phl rd,rs,rt **muleq_s.w.phr rd,rs,rt**	Multiply selected Q15 values from "left"/"right" (higher/lower numbered bits) of *rd*/*rs* to a Q31 result in a general purpose register, Q31-saturating. Like all multiplies which target general purpose registers, it may well use the multiply unit and overwrite *hi*/*lo*, also known as *ac0*.

muleu_s.ph.qbl rd,rs,rt
muleu_s.ph.qbr rd,rs,rt
	A 2×SIMD 16-bit×8-bit multiplication.
	muleu_s.ph.qbl does something like:
	`rd = ((LL_B(rs)*LEFT_H(rt)) << 16) |`
	`      ((LR_B(rs)*RIGHT_H(rt));`
	Note that the multiplications are unsigned integer multiplications, and each half of the result is unsigned-16-bit-saturated.
	The asymmetric source operands are quite unusual, and note this is not a fractional computation.
	muleu_s.ph.qbr is the same but picks the RL and RR (low bit numbered) byte values from *rs*.

mulq_rs.ph rd,rs,rt	2×SIMD Q15 multiplication to two Q15 results. Result in general purpose register, *hi*/*lo* or *ac0* may be overwritten.

Table 9.2 DSP instructions in alphabetical order

Instruction	*Description*
mulsaq_s.w.ph ac,rs,rt	`ac += (LEFT_H(rs)*LEFT_H(rt)) - (RIGHT_H(rs)*RIGHT_H(rt));` The multiplications are done to Q31 values, saturated if they overflow (which is only possible when `-1`¥-1 makes `+1`). The accumulator is really a Q32.31 value, so is unlikely to overflow; no overflow check is done on the accumulation.
packrl.ph rd,rs,rt	pack a “right” and “left” half from different registers, ie `rd = (((rs & 0xFFFF) << 16) | (rt >> 16) & 0xFFFF);`

pick.ph rd,rs,rt	Like a 2-way SIMD conditional move:
	`ph rd,rs,rt;`
	`rd.l = DSPControl[ccond1] ? rs.l: rt.l;`
	`rd.r = DSPControl[ccond0] ? rs.r: rt.r;`

pick.qb rd,rs,rt	Kind of a 4-way SIMD conditional move:
	`qb rd,rs,rt;`
	`rd.ll = DSPControl[ccond3] ? rs.ll: rt.ll;`
	`rd.lr = DSPControl[ccond2] ? rs.lr: rt.lr;`
	`rd.rl = DSPControl[ccond1] ? rs.rl: rt.rl;`
	`rd.rr = DSPControl[ccond0] ? rs.rr: rt.rr;`

preceq.w.phl rd,rt **preceq.w.phr rd,rt**	Convert a Q15 value (either left/high or right/low half of *rt*) to a Q31 value in *rd*.
precequ.ph.qbl rd,rt **precequ.ph.qbla rd,rt** **precequ.ph.qbr rd,rt** **precequ.ph.qbra rd,rt**	Simultaneously convert two unsigned 8-bit fractions from *rt* to Q15 and load into the two halves of *rd*. **precequ.ph.qbl** uses *rt.ll*/*rt.lr*; **precequ.ph.qbla** uses *rt.ll*/*rt.rl*; **precequ.ph.qbr** uses *rt.rl*/*rt.rr*; and **precequ.ph.qbra** uses *rt.lr*/*rt.rr*.
preceu.ph.qbl rd,rt **preceu.ph.qbla rd,rt** **preceu.ph.qbr rd,rt** **preceu.ph.qbra rd,rt**	Zero-extend two unsigned byte values from *rt* to unsigned 16-bit and load into the two halves of *rd*. **preceu.ph.qbl** uses *rt.ll*/*rt.lr*; **preceu.ph.qbla** uses *rt.ll*/*rt.rl*; **preceu.ph.qbr** uses *rt.rl*/*rt.rr*; and **preceu.ph.qbra** uses *rt.lr*/*rt.rr*.
precrq.ph.w rd,rs,rt **precrq_rs.ph.w rd,rs,rt**	**precrq.ph.w** makes a paired-Q15 value by taking the MS bits of the Q31 values in *rs* and *rt*, like this: `rd = (rs & 0xFFFF0000) | ((rt>>16) & 0xFFFF);` **precrq_rs.ph.w** is the same, but rounds and Q15-saturates both half-results.
precrq.qb.ph rd,rs,rt	Form a quad-byte value from two paired-halves. We use the upper 8 bits of each half-word value, as if we were converting an unsigned 16-bit fraction to an unsigned 8-bit fraction. In C: `rd = (rs & 0xFF000000) | (rs<<8 & 0xFF0000) | (rt>>16 & 0xFF00) | (rt>>8 & 0xFF);`
precrqu_s.qb.ph **precrqu_s.qb.ph rd,rs,rt**	Does the same, but each conversion is rounded and saturated to an unsigned byte. Note in particular that a negative Q15 quantity yields a zero byte, since zero is the smallest representable value.
raddu.w.qb rd,rs	Set *rd* to the unsigned 32-bit integer sum of the four unsigned bytes in *rs*.
rddsp rt,mask	Read the contents of the *DSPControl* register into *rt*, but zeroing out any fields for which the appropriate mask bit is zeroed, see Figure 9.1 above.
repl.ph rd,imm **replv.ph rd,rt**	Replicate the same signed value into the two halves of a PH value in *rd*; the value is either provided as an immediate whose range is limited between -512 and +511 (**repl.ph**) or from the *rt* register (**replv.ph**).
repl.qb rd,imm **replv.qb rd,rt**	Replicate the same 8-bit value into all four parts of a QB value in *rd*; the value can come from an immediate constant, or the *rt* register of the **replv.qb** instruction.
shilo ac,shift **shilov ac,rs**	Do a right or left shift (use a negative value for a left shift) of a 64-bit accumulator. The right shift is “logical”, bringing in zeroes into the high bits. **shilo** takes a constant shift amount, while **shilov** get the shift amount from *rs*. The shift amount may be no more than 31 right or 32 left.

Table 9.2 DSP instructions in alphabetical order

Instruction	*Description*
shll.ph rd, rt, sa **shllv.ph rd, rt, rs** **shll_s.ph rd, rt, sa** **shllv_s.ph rd, rt, rs**	2×SIMD (paired-half) shift left. The “**v**” versions take the shift amount from a register, and the “**_s**” versions saturate the result to a signed 16-bit range.
shll.qb rd, rt, sa **shllv.qb rd, rt, rs**	4×SIMD quad-byte shift left, with shift-amount-in-register and saturating (to an unsigned 8-bit result) versions.
shll_s.w rd, rt, sa **shllv_s.w rd, rt, rs**	Signed 32-bit shift left with saturation, with shift-amount-in-register **shllv_s** option.
shra.ph rd, rt, sa **shra_r.ph rd, rt, sa** **shrav.ph rd, rt, rs** **shrav_r.ph rd, rt, rs**	2×SIMD paired-half shift-right arithmetic (“arithmetic” because the vacated high bits of the value are replaced by copies of the input bit 16, the sign bit) - thus performing a correct division by a power of two of a signed number. As usual the **shra_v** variant has the shift amount specified in a register. The **_r** versions round the result first (see the bullet on rounding above).
shra_r.w rd, rt, sa **shrav_r.w rd, rt, rs**	32-bit signed/arithmetic shift right with rounding, see the bullet on rounding.
shrl.qb rd, rt, sa **shrlv.qb rd, rt, rs**	4×SIMD shift right logical (“logical” means that the vacated high bits are filled with zero, appropriate since the byte quantities in a quad-byte are usually treated as unsigned.)
subq.ph rd,rs,rt **subq_s.ph rd,rs,rt**	2×SIMD subtraction. **subq_s.ph** saturates its results to a signed 16-bit range.
subq_s.w rd,rs,rt	32-bit saturating subtraction.
subu.qb rd,rs,rt **subu_s.qb rd,rs,rt**	4×SIMD quad-byte subtraction. Since quad-bytes are treated as unsigned, the saturating variant **subu_s.qb** works to an unsigned byte range.
wrdsp rt,mask	Write the *DSPControl* register with data from *rt*, but leaving unchanged any fields for which the appropriate mask bit is zeroed, see Figure 9.1 above.

9.7 DSP ASE instruction timing

Most DSP ASE operations are pipelined, and instructions can often be issued at the maximum CPU rate, but getting results back into the general-purpose register file takes a few clocks. The timings are generally fairly similar to those for the standard multiply instructions, and are listed - together with delays for the standard instruction set - in Section 6.6.2, "Data dependency delays classified".

Chapter 10

1004K™ CPU features for debug and profiling

In this chapter you'll find:

- Section 10.1, "EJTAG on-chip debug unit"
- Section 10.2 "PDtrace™ instruction trace facility"
- Section 10.3 "CP0 Watchpoints" - monitor code and data access without using EJTAG.
- Section 10.4 "Performance counters" - gather statistics about events, useful for understanding where your program spends its time.

The description here is terse and leaves out some information about EJTAG and PDtrace facilities which are not visible to programmers. We will document it here if it's software visible, or is implementation-dependent information not found in the detailed documentation (see [EJTAG], [PDTRACEUSAGE] and [PDTRACETCB]).

10.1 EJTAG on-chip debug unit

This is a collection of in-CPU resources to support debug. Debug logic serves no direct purpose in the final end-user application, so it's always under threat of being omitted for cost reasons. A debug unit must have virtually no performance impact when not in use; it must use few or no dedicated package pins, and should not increase the logic gate count too much. EJTAG solves the pin issue (and gets its name) by recycling the JTAG pins already included in every SoC for chip test[32].

So the debug unit requires:

- Physical communications with some kind of "probe" device (which is itself controlled by the debug host), achieved through the JTAG pins.
- The ability for a probe to "remote-control" the CPU. The basic trick is to get the CPU to execute instructions that the probe supplies. In turn that's done by directing the CPU to execute code from the magic "dmseg" region where CPU reads and writes are made down the wire to the probe. "dmseg" is itself a part of "dseg", see Section 10.1.6, "The "dseg" memory decode region".
- A distinguished debug exception. In MIPS EJTAG, this is a special "super-exception" marked by a special debug-exception-level flag, so you can use an EJTAG debugger even on regular exception handler code. See Section 10.1.2, "Debug mode" below;
- A number of "hardware breakpoints". Their numerous control registers can't be accommodated in the CP0 register set, so are memory-mapped into "dseg";

[32] It can actually be quite useful to provide EJTAG with its own pins, if your package permits.

- You can take a debug exception from a special breakpoint instruction **`sdbbp`**, on a match from an EJTAG hardware breakpoint, after an EJTAG single-step, when the probe writes the break bit *EJTAG_CONTROL[EjtagBrk]*, or by asserting the external *DINT* (debug interrupt) signal.
- You can configure your hardware to take periodic snapshots of the address of the currently-executing instruction (“PC sampling”) and make those samples available to an EJTAG probe, as described in the next section.

On these foundations powerful debug facilities can be built.

The multi-vendor [EJTAG] specification has many independent options, but MIPS Technologies CPUs tend to have fewer options and to implement the bulk of the EJTAG specification. The 1004K CPU can be configured by your SoC designer with either four instruction breakpoints (or none), and with two data breakpoints (or none). It is also optional whether the dedicated debug-interrupt signal *DINT* is available in your SoC.

10.1.1 Debug communications through JTAG

The chip’s JTAG pins give an external probe access to a special registers inside the CPU. The JTAG standard defines a serial protocol which lets the probe run one of a number of JTAG “instructions”, each of which typically reads/ writes one of a number of registers. EJTAG’s instructions are shown in Table 10.1.

Table 10.1 JTAG instructions for the EJTAG unit

JTAG “Instruction”	*Description*
`IDCODE`	Reads out the MIPS CPU and revision - not very interesting for software, not described further here.
`ImpCode`	Reads bit-field showing what EJTAG options are implemented - see Figure 10.6 below.
`EJTAG_ADDRESS` `EJTAG_DATA`	(read/write) together, allow the probe to respond to instruction fetches and data reads/ writes in the magic “dmseg” region described in Section 10.1.6, "The “dseg” memory decode region".
`EJTAG_CONTROL`	Package of flags and control fields for the probe to read and write; see Figure 10.7 below.
`EJTAGBOOT` `NORMALBOOT`	The “EJTAGBOOT” instruction causes the next CPU reset to lead to CPU booting from probe; see description of the *EJTAG_CONTROL* bits *ProbEn*, *ProbTrap* and *EjtagBrk* in the notes to Figure 10.7. The “NORMALBOOT” instruction reverts to the normal CPU bootstrap.
`FASTDATA`	Special access used to accelerate multi-word data transfers with probe. The probe reads/ writes the 33-bit register formed of a “fast” bit with *EJTAG_DATA*.
`FDC`	Fast Debug Channel. Another accelerated data transfer. This one is accessible by non-debug mode software and it includes FIFOs to separate the software views from the physical data transfer, making it non-blocking. See Section 10.1.11 “Fast Debug Channel”

`TCBCONTROLA`
`TCBCONTROLB`
`TCBCONTROLC`
`TCBCONTROLD`
`TCBCONTROLE`
	Access registers used to control “PDtrace” instruction trace output, if available. See Section 10.2.1 “1004K CPU-specific fields in PDtrace™ JTAG-accessible registers” - only the CPU-specific fields in these registers are documented here.

`PCSAMPLE`	Access register which holds PC sample value, see Section 10.1.15, "PC Sampling with EJTAG".

10.1.2 Debug mode

A special CPU state; the CPU goes into debug mode when it takes any debug exception - which can be caused by an **`sdbbp`** instruction, a hit on an EJTAG breakpoint register, from the external “debug interrupt” signal *DINT*, or single-stepping (the latter is peculiar and described briefly below). Debug mode state is visible as *Debug[DM]* (see Figure 10.1 below). Debug mode (like exception mode, which is similar) disables all normal interrupts. The address map changes in debug mode to give you access to the “dseg” region, described below. Quite a lot of exceptions just won’t happen in debug mode: those which do, run peculiarly - see the relevant paragraphs in Section 10.1.2, "Debug mode".

A CPU with a suitable probe attached can be set up so the debug exception entry point is in the “dmseg” region, running instructions provided by the probe itself. With no probe attached, the debug exception entry point is in the ROM or potentially from an alternate memory location - see Table 7.1.

10.1.3 The debug unit and multi-threading

The software-visible resources of the EJTAG unit are replicated per VPE, and each VPE has its own distinct JTAG “tap”. Just two bits are replicated per-TC: *Debug[SSt]* controls the single-step exception, and *Debug[OffLine]* provides a debugger with a way of controlling exactly which TCs run in between breakpoints of a debug session.

When any TC executes in debug mode, all other TCs (even in other VPEs) are suspended. There is nothing software can do to prevent a debug-mode TC from issuing instructions: it runs regardless of the state of *TCStatus[A]*, *TCHalt*, the *VPEControl[TE]* bit set by **`dmt`**, the *MVPControl[EVP]* bit set by **`dvpe`**, the *VPEConf0[VPA]* bit, or even the debugger’s own *Debug[OffLine]*. However, when you return from debug mode with a **`deret`** and one of these software inhibit bits is active, the TC will not execute any non-debug-mode instruction.

When you execute a debug breakpoint (**`sdbbp`**) instruction or hit a synchronous (address-testing only) breakpoint, the debug exception will be handled by the TC which ran the exception-causing instruction. But an asynchronous entry into debug mode caused by the assertion of *DINT* or hitting a data-testing breakpoint may use any TC affiliated with the VPE which owns the signal or set the breakpoint: and again, this TC is chosen regardless of its software-settable state, so you are guaranteed that the debug condition will be serviced.

When any TC is already executing in debug mode *DINT* (even if directed at another VPE) is ignored.

For non-debug code some MT facilities are protected by “safety catch” control bits. Debug-mode code is all-powerful, as if *VPEConf0[MVP]* was set.

10.1.4 Exceptions in debug mode

Software debuggers will probably be coded to avoid causing exceptions (testing addresses in software, for example, rather than risking address or TLB exceptions).

While executing in debug mode many conditions which would normally cause an exception are ignored: interrupts, debug exceptions (other than that caused by executing **`sdbbp`**), and CP0 watchpoint hits.

But other exceptions are turned into “nested debug exceptions” when the CPU is in debug mode - a facility which is probably mostly valuable to debuggers using the EJTAG probe.

On such a nested debug exception the CPU jumps to the debug exception entry point, remaining in debug mode. The *Debug[DExcCode]* field records the cause of the nested exception, and *DEPC* records the debug-mode-code restart address. This will not be survivable for the debugger unless it saved a copy of the original *DEPC* soon after entering debug mode, but it probably did that! To return from a nested debug exception like this you don’t use **`deret`** (which would inappropriately take you out of debug mode), you grab the address out of *DEPC* and use a jump-register.

10.1.5 Single-stepping

When control returns from debug mode with a **deret** and the (per-TC) single-step bit *Debug[SSt]* is set, the instruction selected by *DEPC* will be executed in non-debug context[33]; then a debug exception will be taken on the program's very next instruction in sequence.

Since at least one instruction is run in normal mode it can lead to a non-debug exception; in that case the "very next instruction in sequence" will be the first instruction of the exception handler, and you'll get a single-step debug exception whose *DEPC* points at the exception handler.

In a multithreaded CPU any number of instructions from other threads might run before you get the single-step exception. A debugger wanting to avoid that can use the various TC's *Debug[OffLine]* controls to inhibit TCs other than the one under debug.

10.1.6 The "dseg" memory decode region

EJTAG needs to use memory space both to accommodate lots of breakpoint registers (too many for CP0) and for its probe-mapped communication space. This memory space pops into existence at the top of the CPU's virtual address map when the CPU is in debug mode, as shown in Table 10.2.

The MIPS trace solution provides software the ability to access the on-chip trace memory. The TCB Registers are mapped to drseg space and this allows software to directly access the on-chip trace memory using load and store instructions.

[33] If *DEPC* points to a branch instruction, both the branch and branch-delay instruction will be executed normally.

Table 10.2 EJTAG debug memory region map ("dseg")

Virtual Address	*Region/sub-regions*		*Location/register*	*Virtual Address*
0xE000.0000	kseg2			0xE000.0000
0xFF1F.FFFF				0xFF1F.FFFF
0xFF20.0000	dseg	dmseg	fastdata	0xFF20.0000
0xFF20.000F				0xFF20.000F
0xFF20.0010				0xFF20.0010
0xFF20.0200			debug entry	0xFF20.0200
0xFF2F.FFFF				0xFF2F.FFFF
0xFF30.0000		drseg	DCR register	0xFF30.0000
0xFF30.0020			DebugVectorAddr	0xFF30.0020
0xFF30.1000			IBS register	0xFF30.1000
			I-breakpoint #0 regs	
0xFF30.1100			IBA0	0xFF30.1100
0xFF30.1108			IBM10	0xFF30.1108
0xFF30.1110			IBASID0	0xFF30.1110
0xFF30.1118			IBC0	0xFF30.1118
			I-breakpoint #1 regs	
0xFF30.1200			IBA1	0xFF30.1200
0xFF30.1208			IBM1	0xFF30.1208
0xFF30.1210			IBASID21	0xFF30.1210
0xFF30.1218			IBC1	0xFF30.1218
			same for next two	
			...	
0xFF30.2000			DBS register	0xFF30.2000
			D-breakpoint #0 regs	
0xFF30.2100			DBA0	0xFF30.2100
0xFF30.2108			DBM10	0xFF30.2108
0xFF30.2110			DBASID0	0xFF30.2110
0xFF30.2118			DBC10	0xFF30.2118
0xFF30.2120			DBV0	0xFF30.2120
0xFF30.2124			DBVHi0	0xFF30.2124
			D-breakpoint #1 regs	
0xFF30.2200			DBA1	0xFF30.2200
0xFF30.2208			DBM1	0xFF30.2208
0xFF30.2210			DBASID1	0xFF30.2210
0xFF30.2218			DBC1	0xFF30.2218
0xFF30.2220			DBV1	0xFF30.2220
0xFF30.2224			DBVHi1	0xFF30.2224
0xFF30.2228				0xFF30.2228
0xFF30.3000			TCB registers	0xFF30.3000
0xFF30.3238				0xFF30.3238
0xFFFF.FFFF				0xFFFF.FFFF

Notes on Table 10.2:

- *dseg*: is the whole debug-mode-only memory area.

  It’s possible for debug-mode software to read the “kseg2”-mapped locations “underneath” by setting *Debug[LSNM]* (see Figure 10.1).

- *dmseg*: is the memory region where reads and writes are implemented by the probe. But if no active probe is plugged in, or if *DCR[PE]* is clear, then accesses here cause reads and writes to be handled like regular “kseg3” accesses.

- *drseg*: is where the debug unit’s main register banks are accessed. Accesses to “drseg” don’t go off CPU. Registers in “drseg” are word-wide, and should be accessed only with 32-bit word-wide loads and stores.

- *fastdata*: is a corner of “dmseg” where probe-mapped reads and writes use a more JTAG-efficient block-mode probe protocol, reducing the amount of JTAG traffic and allowing for faster data transfer. There’s no details about how it’s done in this document, see [EJTAG].

- *debug entry*: is the debug exception entry point. Because it lies in “dmseg”, the debug code can be implemented wholly in probe memory, allowing you to debug a system which has no physical memory reserved for debug.

- *TCB Registers* : These are the PDtrace EJTag Registers. They are physically located in the PDtrace unit, and managed by the PDtrace unit. For software to access the on-chip trace memory, these registers are mapped to drseg.

10.1.7 EJTAG CP0 registers, particularly Debug

In normal circumstances (specifically, when not in debug mode), the only software-visible part of the debug unit is its set of three CP0 registers:

- *Debug* which has configuration and control bits, and is detailed below;

- *DEPC* keeps the restart address from the last debug exception (automatically used by the **deret** instruction);

- *DESAVE* is a CP0 register which is just 32-bits of read/write space. It’s available for a debug exception handler which needs to save the value of a first general-purpose register, so that it can use that register as an address base to save all the others.

Debug, *DEPC* and *DESAVE* are replicated per-VPE, giving each VPE the impression of having its own EJTAG unit.

Debug is the most complicated and interesting. It has so many fields defined that we’ve taken them in three groups: debug exception cause bits in Figure 10.2, information about regular exceptions which want to happen but can’t because you’re in debug mode in Figure 10.3, and everything else. The "everything else" category includes the most important fields and comes first, in Figure 10.1.

Figure 10.1 Fields in the EJTAG CP0 Debug register

31	30	29	28	27	26	25	24 21	20	19 18	17 15	14 10	9	8	7	6	5 0
DBD	DM	NoDCR	LSNM	Doze	Halt	Count DM	*pending* (Figure 10.3)	IE XI	*cause* (Figure 10.2)	EJTAGver	DExc Code	NoSSt	SSt	OffLine	0	*cause* (Figure 10.2)

These fields are:

DBD: exception happened in branch delay slot. When this happens *DEPC* will point to the branch instruction, which is usually the right place to restart.

DM: debug mode - set on debug exception from user mode, cleared by **`deret`**.

Then some configuration and control bits:

NoDCR: read-only - 0 if there is a memory-mapped *DCR* register. MIPS Technologies CPUs will always have one. Any EJTAG unit implementing "dseg" at all implements *DCR*.

LSNM: Set this to 1 if you want debug-mode accesses to "dseg" addresses to be just sent to system memory. This makes most of the EJTAG unit's control system unavailable, so will probably only be done around a particular load/store.

Doze: before the debug exception, CPU was in some kind of reduced power mode.

Halt: before the debug exception, the CPU was stopped - probably asleep after a **`wait`** instruction.

CountDM: 1 if and only if the count register continues to run in debug mode. Writable for the 1004K CPU, so you get to choose. On some other implementations it's read-only and just tells you what the CPU does.

IEXI: set to 1 to defer imprecise exceptions. Set by default on entry to debug mode, cleared on exit, but writable. The deferred exception will come back when and if this bit is cleared: until then you can see that it happened by looking at the "pending" bits shown in Figure 10.3 below.

EJTAGver: read-only - tells you which revision of the specification this implementation conforms to. On the 1004K CPU it reads 5 for version 5.0. The full set of legal values are:

0	Version 2.0 and earlier
1	Version 2.5
2	Version 2.6
3	Version 3.1
4	Version 4.0
5	Version 5.0

DExcCode: Cause of any non-debug exception you just handled from within debug mode - following first entry to debug mode, this field is undefined. The value will be one of those defined for *Cause[ExcCode]*, as shown in Table C.4.

NoSSt: read-only - reads 0 because single-step is implemented (it always is on MIPS Technologies CPUs).

SSt: set 1 to enable single-step.

OffLine: prevents a TC from running any instructions (except in debug mode, but then debug mode overrides *all* software inhibitions on thread scheduling). It's there for debuggers which may need to selectively stop some threads, and should not be used by application or OS code. This bit has to be replicated per-TC. often not implemented in non-MT CPUs.

Figure 10.2 Exception cause bits in the debug register

	31 20	19	18	17 6	5	4	3	2	1	0
Debug		DDBSImpr	DDBLImpr		DINT	DIB	DDBS	DDBL	DBp	DSS

DDBSImpr: imprecise store breakpoint - see Section 10.1.14, "Imprecise debug breaks" below. *DEPC* probably points to an instruction some time later in sequence than the store which triggered the breakpoint. The debugger or user (or both) have to cope as best they can.

DDBLImpr: imprecise load breakpoint. (See note on imprecise store breakpoint, above).

DINT: debug interrupt: either the *DINT* signal got asserted or the probe wrote *EJTAG_CONTROL[EjtagBrk]* through the JTAG signals.

DIB: instruction breakpoint. If *DBp* is clear, that must have been from an **`sdbbp`**.

DDBS: precise store breakpoint.

DDBL: precise load breakpoint.

DBp: any sort of match with a hardware breakpoint.

DSS: single-step exception.

Figure 10.3 Debug register - exception-pending flags

	31 25	24	23	22	21	20 0
Debug		IBusEP	MCheckP	CacheEP	DBusEP	

These note exceptions caused by instructions run in debug mode, but which have not happened yet because they are imprecise and *Debug[IEXI]* is set. They remain set until *Debug[IEXI]* is cleared explicitly or implicitly by a **`deret`**, when the exception is delivered and the pending bit cleared:

IBusEP: bus error on instruction fetch pending. This exception is precise on the 1004K CPU, so this can't happen and the field is always zero.

MCheckP: machine check pending (usually an illegal TLB update). As above, machine check exceptions are not generrated on the 1004K CPU, so this is always zero.

CacheEP: cache parity error pending.

DBusEP: bus error on data access pending.

10.1.8 The DCR (debug control) memory-mapped register

This is a memory-mapped EJTAG register . It's found in "drseg" at location 0xFF30.0000 as shown in Table 10.2 (but only accessible if the CPU is in debug mode). The fields are in Figure 10.4:

Figure 10.4 Fields in the memory-mapped DCR (debug control) register

31	30	29	28	27	26	25	24	23	22	21		19	18	17	16
Res		ENM	Res		PCIM	PCno ASID	DASQ	DASe	DAS				FDCI mpl	DB	IB

15	14	13	12	11	10	9	8		6	5	4	3	2	1	0
IVM	DVM	0		RDVec	CBT	PCS	PCR			PCSE	INTE	NMIE	NMIP	SRE	PE

Where:

ENM: (read only) reports CPU endianness (1 == big).

FDCImpl: (read only) 1 if the Fast Debug Channel is available. See Section 10.1.11 "Fast Debug Channel" for details

DB/IB: (read only) 1 if data/instruction hardware breakpoints are available, respectively. The 1004K CPU has either 0 or 2 data breakpoints, and either 0 or 4 instruction breakpoints.

IVM: (read-only) tells you if an inverted data value match on data hardware breakpoints is implemented.

DVM: (read-only) tells you if a data value store on a data value breakpoint match is implemented.

RdVec: If set, use the address specified in *DebugVectorAddr* register for debug exceptions instead of the default ROM address. If the probe is handling debug exceptions, it will continue to take precedence over this.

CBT: (read-only) tells you if a complex breakpoint block is implemented.

PCS, PCR: *PCS, PCSE,PCIM,PCnoASID: PCS* reads 1 if the PC sampling feature is available, as it can be on the 1004K CPU. Then *PCSE* enables PC sampling and *PCR* is a three-bit field defining the sampling frequency as one sample every $2^{(5+\mathrm{PCR})}$ cycles. *PCnoASID* indicates or controls whether the ASID field is included in the sample. *PCIM*, if settable, enables only sampling the PC of instructions that missed in the instruction cache. See Section 10.1.15, "PC Sampling with EJTAG" for details.

DAS, DASQ, DASE: DAS reads 1 if the Data Address Sampling feature is available. If supported, this feature builds on top of the PC sampling mechanisms to sample data addresses. *DASE* enables DAsampling and is not mutually exclusive with *PCSE*. *DASQ* limits the data address samples to those addresses that match on data breakpoint 0.

INTE/NMIE: set *DCR[INTE]* zero to disable interrupts in non-debug mode (it's a separate bit from the various non-debug-mode visible interrupt enables). The idea is that the debugger might want to step through kernel code or run kernel subroutines (perhaps to discover OS-related information) without losing control because interrupts start up again.

DCR[NMIE] masks non-maskable interrupt in non-debug mode (a nice paradox). Both bits are "1" from reset.

NMIP: (read-only) tells you that a non-maskable interrupt is pending, and will happen when you leave debug mode (and according to *DCR[NMIE]* as above).

SRE: if implemented, write zero to mask soft-reset causes. This signal has no effect inside the 1004K CPU but is presented at the interface, where customer reset logic could be influenced by it.

PE: (read only) software-readable version of the probe-controlled enable bit *EJTAG_CONTROL[ProbEn]*, which you could look at in Figure 10.7.

10.1.9 The DebugVectorAddr memory-mapped register

This is another memory-mapped EJTAG register . It’s found in “drseg” at location 0xFF30.0020 as shown in Table 10.2 (but only accessible if the CPU is in debug mode). The fields are in Figure 10.5:

If enabled via the *RdVec* bit in the *DCR*, this register will control the address used for debug exceptions when a debug probe is not handling them. By default, the exception handler is located in the boot ROM. If the debugger is allowed some space in RAM, it can both customize the debug handler and execute faster than from ROM. You can even make the handler cacheable to speed it up further - of course with the penalty of altering the cache behavior of the program you are debugging. In some cases, that tradeoff may be useful

Figure 10.5 Fields in the memory-mapped DCR (debug control) register

31 30	29 7	6 1	0
10	DebugVectorOffset	0	ISA

Where:

DebugVectorOffset: Specify the intermediate bits of the desired debug exception vector. The upper bits of the vector are fixed to restrict it to kseg0 or kseg1. The lower bits of the vector are fixed to save some hardware costs for no real loss in functionality.

ISA: In cores with the microMIPS ISA, this bit can specify which ISA the exception handler is built in. This is tied to 0 on this core as the MIPS16 ASE does not have the privledged operations that would make it useful as an exception handler.

10.1.10 JTAG-accessible registers

We’re wandering away from what is relevant to software here: these registers are available for read and write only by the probe, and are not software-accessible.

But you can’t really understand the EJTAG unit without knowing what dials, knobs and switches are available to the probe, so it seems easier to give a little too much information.

First of all there are two informational fields provided to the probe, *IDCODE* (just reflects some inputs brought in to the CPU by the SoC team, not very interesting) and *ImpCode* (Figure 10.6); then there’s the main CPU interaction control/status register *EJTAG_CONTROL* (Figure 10.7).

Figure 10.6 Fields in the JTAG-accessible ImpCode register

31 29	28 25	24	23 21	20 17	16	15	14	13 1	0
EJTAGver 5 = 5.0	0	DINTsup *see note*	ASIDsize *see note*	0	MIPS16 1	0	NoDMA 1	0	MIPS32/64 0

Notes on the *ImpCode* fields:

EJTAGver: same value (and meaning) as the *Debug[EJTAGver]* field, see the notes on Figure 7-2.

DINTsup: whether JTAG-connected probe has a *DINT* signal to interrupt the CPU. Configured by your SoC designer (who should know) by hard-wiring the CPU interface signal *EJ_DINTsup*.

The probe can always interrupt the CPU by a JTAG command using the *EJTAG_CONTROL[EjtagBrk]*, but *DINT* is much faster, which is useful if you're cross-triggering one piece of hardware from another. However, it is fed to both VPEs at once, and it's unpredictable which of them will take the resulting debug exception (only one can).

ASIDsize: usually 2 (indicating the 8-bit *EntryHi[ASID]* field size required by the MIPS32 standard), but can be 0 if your CPU has been built with the no-TLB option (i.e. a fixed-mapping MMU).

MIPS16: 1 because the 1004K CPU always supports the MIPS16 instruction set extension.

NoDMA: 1 - MIPS Technologies CPUs do not provide EJTAG "DMA" (which would allow a probe to directly read and write anything attached to the 1004K CPU's OCP interface).

MIPS32/64: the zero indicates this is a 32-bit CPU.

Rocc: "reset occurred" - reads 1 while a reset signal is applied to the CPU - and then the 1 value persists until overwritten with a zero from the JTAG side. Until the probe reads this as zero most of the other fields are nonsense.

The EJTAG_CONTROL register is shown in Figure 10.7:

Table 10.3 Fields in the JTAG-accessible EJTAG_CONTROL register

31	30 29	28 24	23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	Res		Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	Res	DM	Res

Figure 10.7 Fields in the JTAG-accessible EJTAG_CONTROL register

31	30 29	28 23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	0	Doze	Halt	PerRst	PRnW	PrAcc	0	PrRst	ProbEn	ProbTrap	0	EjtagBrk	0	DM	0

Notes on the fields:

Rocc: (read/write) is 1 when a CPU reset has occurred since the bit was last cleared. The *Rocc* bit will keep the 1 value as long as reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected. The *EJTAG Control* register is not updated in the *Update-DR* state unless *Rocc* is 0, or written to 0. This is in order to ensure proper handling of processor access.

Psz: (read-only) when software reads or writes "dmseg" this tells the probe whether it was a word, byte or whatever-size transfer:

Byte-within-word address	*Size code*	*Transfer Size*
EJTAG_ADDRESS[1-0]	EJTAG_CONTROL[Psz]	
X	0	Byte
00	1	Halfword
10		
00	2	Word
00	3	Tri-byte (lowest address 3 bytes)
01		Tri-byte (highest address 3 bytes)

Doze/Halt: (read-only) indicates CPU not fully awake. *Doze* reflects any reduced-power mode, whereas *Halt* is set only if the CPU is asleep after a **`wait`** or similar.

PerRst: write to set the *EJ_PerRst* output signal from the CPU, which can be used to reset non-CPU logic (ask your SoC designer whether it's connected to anything).

For this and all other fields which change CPU state, we recommend that the probe should write the field and then poll for the change to be reflected in this register, which may take a short while. In some cases the bit is just an output one, when the readback will be pointless (but harmless).

PRnW/PrAcc: *PrAcc* is 1 when the CPU is doing a read/write of the "dmseg" region, and the probe should service it. The "slow" read/write protocol involves the probe flipping this bit back to zero to tell the CPU the transfer is ready.

While *PrAcc* is active the read-only *PRnW* bit distinguishes writes (1) from reads (0).

PrRst: controls the *EJ_PrRst* signal from the CPU, which may be wired back to reset the CPU and related logic. Write a 1 to reset. If it works, the probe will eventually see the bit fall back to 0 by itself, as the CPU resets. Most probes are wired up with a direct CPU reset signal, which is more reliable.

ProbEn, ProbTrap, EjtagBrk: *ProbEn* must be set before CPU accesses to "dmseg" will be sent to the probe. It can be written by the probe directly. *ProbTrap* relocates the debug exception entry point from 0xBFC0.0480[34] (when 0) to the "dmseg" location 0xFF20.0200 - required when the debug exception handler itself is supplied by the probe.

EjtagBrk can be written 1 to "interrupt" the CPU into debug mode.

The three come together into a trick to support systems wanting to boot from EJTAG. The value of all these three bits is preset by the "EJTAGBOOT" JTAG instruction. When the CPU resets with all of these set to 1, then the CPU will immediately enter debug mode and start reading instructions from the probe.

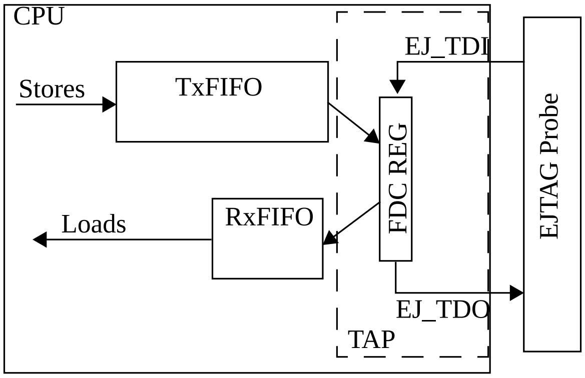
DM: (read-only) indicates the CPU is in debug mode, a probe-readable version of *Debug[DM]*.

10.1.11 Fast Debug Channel

The Fast Debug Channel (or FDC) is an interesting creature. It provides a mechanism for data transfers between the probe and the CPU, but unlike some of the other mechanisms of that type, it is not constrained to debug mode access. Kernel mode software can access the memory mapped interface and can even grant access rights to user or supervisor programs. The memory mapped registers provide basic configuration, status, and control information as well as giving access to the transmit (CPU to probe) and receive FIFOs. These FIFOs are included to isolate the software visible interface from the physical transfer of bits to the probe and allow some 'burstiness' of data. Associated with each 32-bit piece of data is a 4-bit Channel ID. Figure 10.8 shows a high level view of the data paths.

[34] The ROM-exception-area debug entry point can be relocated by hardware option, see Table 7.1 and its notes.

Figure 10.8 Fast Debug Channel



The memory mapped registers are part of the Common Device Memory Map, see Section 5.7 “Common Device Memory Map” for details. Table 10.4 shows the address offsets of the FDC registers within the device block.

Table 10.4 FDC Register Mapping

Offset in CDMM device block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n (0 ≤ n ≤ 15)

Each device within the CDMM begins with an Access Control and Status Register which gives information about the device and also provides a means for giving user and supervisor programs access to the rest of the device. The *FDACSR* is shown in Figure 10.9

Figure 10.9 Fields in the FDC Access Control and Status (FDACSR) Register

31 24	23 22	21 16	15 12	11 4	3	2	1	0
DevID	zero	DevSize	DevRev	zero	Uw	Ur	Sw	Sr

Where:

DevID: (read only) indicates the device ID - 0xfd in this case.

DevSize: (read only) indicates how many 64B blocks (minus 1) this device uses - value of 2, indicating 3 blocks for FDC

DevRev: (read only) Revision number of the device - currently 0.

Uw/Ur: control whether write and reads, respectively, from user programs are allowed to access the device registers. If 0, reads will return 0 and writes will be dropped.

Sw/Sr: Same idea as *Uw/Ur*, but for supervisor access

The *FDCFG* register gives some configuration information and allows software to specify if and when FDC interrupts are to be generated. The interrupt threshholds can be adjusted for different aims: no interrupts, minimzing the CPU overhead by allowing the CPU to completely fill or drain the FIFO with one interrupt, maximizing bandwidth by interrupting slightly earlier to avoid wasting transfers of null transmit data or non accepted receive data, or minimum latency to be interrupted as soon as data is available. This register is shown in Figure 10.10

Figure 10.10 Fields in the FDC Config (FDCFG) Register

31 20	19 18	17 16	15 8	7 0
0	TxIntThresh	RxIntThresh	TxFIFOSize	RxFIFOSize

Where:

TxIntThresh: Controls when an interrupt is generated based on the occupancy of the transmit FIFO
0 - Interrupts Disabled
1 - FIFO empty (minimum CPU overhead)
2 - FIFO not full
3 - Almost empty - 0 or 1 entries in use (maximum bandwidth)

RxIntThresh: Controls when an interrupt is generated based on the occupancy of the receive FIFO
0 - Interrupts Disabled
1 - FIFO full (minimum CPU overhead)
2 - FIFO not empty (minimum latency)
3 - Almost full - 0 or 1 entries in use (maximum bandwidth)

Tx/RxFIFOSize: (read only) indicates how many entries are in each FIFO

The *FDSTAT* register is a read-only register that gives the current status of the FIFOs. The fields are shown in Figure 10.11.

Figure 10.11 Fields in the FDC Status (FDSTAT) Register

31 24	23 16	15 8	7 4	3	2	2	0
TxCount	RxCount	0	RxChan	RxE	RxF	TxE	TxF

Where:

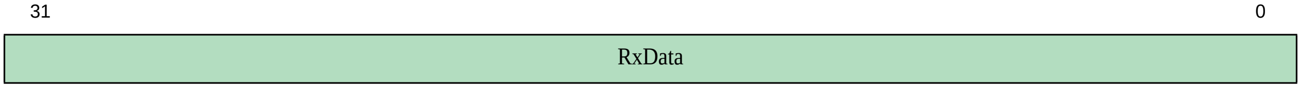
Tx/RxCount: Optional fields indicating how many FIFO entries are in use. These fields are not implemented and will read as 0

RxChan: Channel Identifier for the receive data at the head of the RxFIFO. Not meaningful if *RxE*==1

RxE/RxF/TxE/TxF: Status of each FIFO. Each one can be either Empty, Full, or somewhere in the middle, in which case neither E nor F would be set.
TxF must be checked prior to attempting a write to the transmit FIFO
RxE must be checked prior to attempting a read from the receive FIFO
The other two status bits would not generally be as useful, but are provided for symmetry

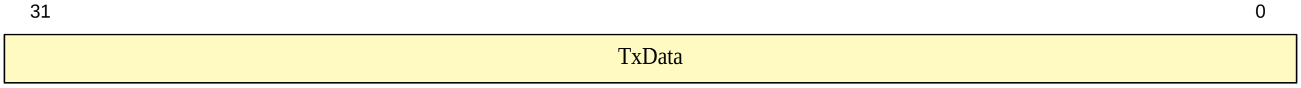
The *FDRX* register is a read-only register that returns the top entry in the receive FIFO. It is undefined if *FDSTAT[RxE]*==1, so that register should be checked prior to reading. That check will also return the ChannelID so you know what type of data this is.

Figure 10.12 Fields in the FDC Receive (FDRX) Register



The *FDTXn* registers are 16 write-only registers that write into the bottom entry in the transmit FIFO. The 16 copies provide the means for selecting a ChannelID for the write data. The address used for the write is decoded into the 4-bit ChannelID and written into the FIFO with the data. Results are undefined if *FDSTAT[TxF]*==1, so that register should be checked prior to writing data.

Figure 10.13 Fields in the FDC Transmit (FDTXn) Registers



10.1.12 EJTAG breakpoint registers

It's optional whether the 1004K CPU has EJTAG breakpoint registers. It can have up to 4 instruction breakpoints and up to 2 data breakpoints. The breakpoints:

- Work only on virtual addresses, not physical addresses. However, you can restrict the breakpoint to a single address space by specifying an "ASID" value to match. Debuggers will need the co-operation of the OS to get this right.

- Use a bit-wise address mask to permit a degree of fuzzy matching.

- On the data side, you can break only when a particular value is loaded or stored. However, such breakpoints are imprecise in a CPU like the 1004K CPU - see Section 10.1.14, "Imprecise debug breaks" below.

There are instruction-side and data-side breakpoint status registers (they're located in "drseg", accessible only when in debug mode, and their addresses are in Section 10.2, "EJTAG debug memory region map ("dseg")".) They're called *IBS* and *DBS*. The latter has, in theory, two extra fields (bits 29-28) used to flag implementations which can't do a load/store break conditional on the data value. However, MIPS CPUs with hardware breakpoints always include the value check, so these bits read zero anyway. So the registers are as shown in Figure 10.14.

Figure 10.14 Fields in the IBS/DBS (EJTAG breakpoint status) registers

	31	30	29 28	27 24	23 4	3 2	1 0
DBS	0	ASID-sup	0	BCN = 2	0		BS1-0
IBS				BCN = 4	0	BSD3-0	

Where:

ASIDsup: is 1 if the breakpoints can use ASID matching to distinguish addresses from different address spaces; on the 1004K CPU that's available if and only if a TLB is fitted.

BCN: the number of hardware breakpoints available (up to two data, up to four instructions).

BS1-0, BSD3-0: bitfields showing breakpoints which have been matched. Debug software has to clear down a bit after a breakpoint is detected.

Then each EJTAG hardware breakpoint ("n" is 0-3 to select a particular breakpoint) is set up through 4-6 separate registers:

- *IBCn, DBCn*: breakpoint control register shown at Figure 7-9 below;

- *IBAn, DBAn*: breakpoint address;

- *IBAMm, DBAMn*: bitwise mask for breakpoint address comparison. A "1" in the mask marks an address bit which will be *excluded from* comparison, so set this zero for exact matching.

  Ingeniously, *IBAMm[0]* corresponds to the slightly-bogus instruction address bit zero used to track whether the CPU is running MIPS16 instructions, and allows you to determine whether an EJTAG I-breakpoint may apply only in MIPS16 (or non-MIPS16) mode.

- *IBASIDn, DBASIDn* specifies an 8-bit ASID, which may be compared against the current *EntryHi[ASID]* field to filter breakpoints so that they only happen to a program in the right "address space". The ASID check can be enabled or disabled using *IBCn[ASIDuse]* or *DBCn[ASIDuse]* respectively - see Figure 7-9 and its notes below. ID (so that the break will only affect one Linux process, for example).

  The higher 24 bits of each of these registers is always zero.

- *DBVn, DBVHin* the value to be matched on load/store breakpoints. *DBCHin* defines bits 63-32 to be matched for 64-bit load/stores: the 32-bit[35] 1004K has 64-bit load/store instructions for floating point.

  Note that you can disable data matching (to get an address-only data breakpoint) by setting the value byte-lane comparison mask *DBCn[BLM]* to all 1s.

So now let's look at the control registers in Figure 10.15.

[35] A JTAG hardware breakpoint for a real 64-bit CPU would have 64-bit *DBVn* registers, so wouldn't need *DBVHin*.

Figure 10.15 Fields in the hardware breakpoint control registers (IBCn, DBCn)

	31 24	23	22 18	17 14	13	12	11 8	7 4	3	2	1	0
DBCn	0	ASIDuse	0	BAI7-0	NoSB	NoLB	0	BLM7-0	0	TE	0	BE

	31 24	23	22 3	2	1	0
IBCn	0	ASIDuse	0	TE	0	BE

The fields are:

ASIDuse: set 1 to compare the ASID as well as the address.

BAI7-0: "byte (lane) access ignore"[36] - which sounds mysterious. But this is really an *address* filter.

When you set a data breakpoint, you probably want to break on any access to the data of interest. You don't usually want to make the break conditional on whether the access is done with a load byte, load word, or even load-word-left: but the obvious way of setting up the address match for a breakpoint has that effect.

To make sure you catch any access to a location, you can use the address mask to disable sub-doubleword address matching and then use *DBCn[BAI]* to mark the bytes of interest inside the doubleword: well, except that zero bits mark the bytes of interest, and 1 bits mark the bytes to ignore (hence the mnemonic).

The *DBCn[BAI]* bits are numbered by the byte-lane within the 64-bit on-chip data bus; so be careful, the relationship between the byte address of a datum and its byte lane is endianness-sensitive.

NoSB, NoLB: set *0 to enable*[37] breakpoint on store/load respectively.

BLM7-0: a per-byte mask for data comparison. A zero bit means compare this byte, a 1 bit means to ignore its value. Set this field all-ones to disable the data match.

TE: set 1 to use as trigger for "PDtrace" instruction tracing as described in Section 10.2 "PDtrace™ instruction trace facility" below.

BE: set 1 to activate breakpoint. This fields resets to zero, to avoid spurious breakpoints caused by random register settings: don't forget to set it!

10.1.13 Understanding breakpoint conditions

There are a lot of different fields and settings which are involved in determining when a hardware breakpoint detects its condition and causes an exception.

In all cases, there will be no break if you're in debug mode already... but then for a break to happen:

- *For all breakpoints including instructions:* all the following must be true:

[36] Why are there 8 bytes, when the 1004K CPU is a 32-bit CPU with only 32-bit general purpose registers? Well, the *DBCn[BAI]* and *DBCn[BLM]* fields each have a bit for each byte-lane across the data bus, and the 1004K CPU has a 64-bit data bus (and in fact can do 64-bit load and store operations, for example for floating point values).

[37] "1-to-enable" would feel more logical. The advantage of using 0-to-enable here is that the zero value means "break on either read or write", which is a better default than "never break at all".

1. The breakpoint control register enable bit *IBAn[BE]*/*DBAn[BE]* is set.

2. the address generated by the program for instruction fetch, load or store matches those bits of the breakpoint's address register *IBAn*/*DBAn* for which the corresponding address-mask register bits in *IBAn*/*DBAn* are zero.

3. either *IBCn[ASIDuse]*/*DBCn[ASIDuse]* is zero (so we don't care what address space we're matching against), OR the address-space ID of the running program, i.e. *EntryHi[ASID]*, is equal to the value in *IBASIDn*/ *DBASIDn*.

That's all for instruction breakpoints, but for data-side breakpoints also:

- *Data compare break conditions (not value related)*: both the following must be true:

4. It's a load and *DBCn[NoLB]* is zero, or it's a store and *DBCn[NoSB]* is zero.

5. The load or the store touches at least one byte-within-doubleword for which the corresponding *DBCn[BAI]* bit is zero.

If you didn't want to compare the load/store value then *DBCn[BLM]* will be all-ones, and you're done. But if you also want to consider the value:

- *Data value compare break conditions*:

6. the data loaded or stored, as it would appear on the system bus, matches the 64-bit contents of *DBVHin* with *DBVn* in each of those 8-bit groups for which the corresponding bit in *DBCn[BLM]* is zero.

That's it.

10.1.14 Imprecise debug breaks

Instruction breakpoints, and data breakpoints filtering only on address conditions are *precise*; that means that:

1. *DEPC* will point at the fetched or load/store instruction itself (except if it's in a branch delay slot, will point at the branch instruction);

2. The instruction will not have caused any side effects; in particular, the load/store will not reach the cache or memory.

Most exceptions in MIPS architecture CPUs are precise. But because of the way the 1004K CPU optimizes loads and stores by permitting the CPU to run on at least until it needs to use the data from a load, data breakpoints which filter on the data value are *imprecise*. The debug exception will happen to whatever instruction (typically later in the instruction stream) is running when the hardware detects the match, and not necessarily to the same TC. The debugging software must cope.

10.1.15 PC Sampling with EJTAG

A valuable trick available with recent revisions of the EJTAG specification and probes, "PC sampling" provides a non-intrusive way to collect statistical information about the activity of a running system. You can tell whether PC sampling is enabled by looking at *DCR[PCS]*, as shown in Figure 7-5 above.

The hardware snapshots the "current PC" periodically, and stores that value where it can be retrieved by a debug probe. It's then up to software to construct a histogram of samples over a period of time, which (statistically) allows a programmer to see where the CPU has spent most cycles. Not only is this useful, but it's also familiar: systems have used intrusive interrupt-based PC-sampling for many years, so there are tools which can readily interpret this sort of data.

When PC sampling is configured in your CPU, it runs continuously. Some sleight of hand is used if the CPU is hanging on a **`wait`** instruction. Rather than wasting even a small amount of power running the counter and resampling the PC of the **`wait`** instruction, the hardware simply keeps the "new" bit set while it is in this state telling the profiling software that yes, we are still at that instruction. You can choose to sample as often as once per 32 cycles or as rarely as once per 4096 cycles[38]; at every sampling point the address of the instruction completing in that cycle (or if none completes, the address of the next instruction to complete) is deposited in a JTAG-accessible register. Sampling rate is controlled by the *DCR[PCR]* field of the debug control register shown in Figure 7-5.

In addition to the 32 bits of the instruction address, several other fields are stored by the hardware to help identify the instruction. The aforementioned "new" bit indicates a new sample, which a probe can use to avoid double-counting the same sample. On multi-threaded CPUs where there might be several copies of the code running, a TCID field is also appended. The then-current ASID may also be included so that you can interpret the virtual PC. The ASID is included unless the *DCR[PCnoASID]* bit is set. This bit may be hardwired in a given implementation or the bit might be writable, so go ahead and try to change it if you feel like it (but be sure to read it back and see if the write 'stuck' so that you know how many bits to scan and how to interpret them).

Later versions of the EJTAG specification have made further extensions to PC sampling to enable some different types of analysis. As of this writing, these modes were not implemented, but they may be added at some point. If the *DCR[PCIM]* bit is settable, it will restrict PC sampling to only report PCs of instructions that missed in the instruction cache. This can highlight code segments that cause contention where some fine tuning of code alignment may yield improved performance. The other major extension is data address (DA) sampling whose presence is indicated by the *DCR[PCS]* bit. The enables the same sort of profiling on load and store addresses that PC sampling enables on code addresses. To focus in on a specific subset of accesses, a data hardware breakpoint can be used to qualify which ones are sampled. DA and PC sampling are enabled independently and can be active simultaneously (of course the two samples share the same serial scan path, so enabling both reduces the number of samples of either)

10.1.16 JTAG-accessible and memory-mapped PDtrace TCB Registers

The DCR and the hardware breakpoint registers are EJTAG registers that are both JTAG-accessible and memory-mapped. In addition to the DCR and the hardware breakpoint registers, the EJTAG PDtrace Registers listed in Table 10.5 are also memory-mapped to drseg. These registers allow software to access the on-chip trace memory. A load from the EJAG register *TCBTW* will return the data at the address location pointed to by the read pointer *TCBRDP*. See the *Software User's Manual* for more details and rules to access the on-chip trace memory.

Table 10.5 Mapping TCB Registers in drseg

Offset in drseg	Register Name	Description
0x3000	*TCBControlA*	The TCBControlA register.
0x3008	*TCBControlB*	The TCBControlB register.
0x3010	*TCBControlC*	The TCBControlC register.

[38] Since it runs continuously, it's a good thing that from reset the sampling period defaults to its maximum.

Table 10.5 Mapping TCB Registers in drseg (Continued)

Offset in drseg	Register Name	Description
0x3018	*TCBControlD*	The TCBControlD register.
0x3020	*TCBControlE*	The TCBControlE register.
0x3028	*TCBConfig*	The TCBConfig register.
0x3100	*TCBTW*	Trace Word read register. This register holds the Trace Word just read from on-line trace memory.
0x3108	*TCBRDP*	Trace Word Read pointer. It points to the location in the on-line trace memory where the next Trace Word will be read. A TW read has the side-effect of post-incrementing this register value to point to the next TW location. (A maximum value wraps the address around to the beginning of the trace memory).
0x3110	*TCBWRP*	Trace Word Write pointer. It points to the location in the on-line trace memory where the next new Trace Word will be written.
0x3118	*TCBSTP*	Trace Word Start Pointer. It points to the location of the oldest TW in the on-chip trace memory.
0x3120	*BKUPRDP*	This is not a TCB register, but needed on a reset to save the *TCBRDP* value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of TCBRDP before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3128	*BKUPWRP*	This is not a TCB register, but needed on a reset to save the *TCBWRP* value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of *TCBWRP* before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3130	*BKUPSTP*	This is not a TCB register, but needed on a reset to save the TCBSTP value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of *TCBSTP* before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3200-0x3238	*TCBTrigX*	The *TCBTrigX* set of registers. The number of implemented registers is determined by the value in $\mathit{TCBCONFIG}_{\mathrm{TRIG}}$.

10.2 PDtrace™ instruction trace facility

An instruction trace is a set of data generated when a program runs which allows you to recreate the sequence of instructions executed, possibly with additional information included about data values. Instruction traces rapidly become enormous, and are typically generated in some kind of abbreviated form, which may be reconstructed by software which is in possession of a copy of the binary code of your system.

1004K family CPUs can be configured with PDtrace logic, which provides a non-intrusive way of finding out what instructions your CPU ran. If your system includes PDtrace logic, *Config3[TL]* will read 1.

With a very high-speed CPU like the 1004K CPU this is challenging, because you need to send data so fast. The PDtrace system deals with this by:

- *Compressing the trace*: a software tool in possession of the binary of your program can predict where execution will go next, following sequential instructions and fixed branches. To trace your program it needs only to know whether conditional branches were taken, and the destination of computed branches like jump-register.

- *Switching the trace on and off*: the 1004K CPU can be configured with up to 8 “trace triggers”, allowing you to start and stop tracing based on EJTAG breakpoint matches: see Section 10.1.12, "EJTAG breakpoint registers" above and Table 10.21 below.

- *High-speed connection to a debug/trace probe*: optional. But if fitted, it uses advanced signalling techniques to get trace data from the CPU CPU, out of dedicated package pins to a probe. Good probes have generous amounts of high-speed memory to store long traces.

  TraceControl2[ValidModes,TBI,TBU] (described below at Figure 7-10 and following) tell you whether you have such a connection available on your CPU. You’ll have to ask the hardware engineers whether they brought out the connector, of course.

- *Very high-speed on-chip trace memory*: if fitted, you may find between 256bytes and 8Mbytes of trace memory in your system (larger than a few Kbytes is unlikely). Again, see *TraceControl2[ValidModes,TBI,TBU]* to find out what facilities you have.

- *Option to slow the CPU to match the tracing speed*: when you really, really need a full trace, and are prepared to slow down your program if necessary to wait while the trace information is sent to the probe. This is controlled by *TraceControl[IO]*, see below.

- *Software access to on-chip trace memory* : A new mechanism is provided to allow software to read the on-chip trace memory. This is achieved by mapping all the TCB registers to drseg.

In practice the PDtrace logic depends on the existence of an EJTAG unit (described in the previous section) and an enhanced EJTAG probe. To benefit from on-probe trace memory, the probe will need to attach to PDtrace-specific signals.

This manual describes only the lowest-level building blocks as visible to software. For real hardware information refer to [PDTRACETCB]; for [PDTRACETCB]guidance about how to use the PDtrace facilities for software development see [PDTRACEUSAGE]. To use PDtrace facilities, you’ll have to read the software manuals which come with a probe.

10.2.1 1004K CPU-specific fields in PDtrace™ JTAG-accessible registers

The PDtrace system is controlled by the JTAG-accessible registers TCBCONTROLA, TCBCONTROLB, TCBCONTROLC, TCBCONTROLD, and TCBCONTROLE. Normally they are not visible to software running on the CPU, but we'll document fields and configured values which are specific to 1004K family CPUs. With the new feature of enabling software to access the on-chip trace memory, all the JTAG-accessible registers are visible to software via a load or store to their drseg memory mapped location.

Table 10.6 Fields in the TCBCONTROLA register

31 30	29 27	26	25 24	23	22 20	19	18	17	16	15	14	13	12 5	4	3	2	1	0
SyPExt	Impl	0	VModes	ADW	SyP	TB	IO	D	E	S	K	U	ASID	G	TFCR	TLSM	TIM	On

In TCBCONTROLA:

VModes: reads "1 0", showing that 1004K family CPUs support all tracing modes.

ADW: reads "1" to indicate that we support the wide (32-bit) internal trace bus.

Ineff: set to 1 to indicate that CPU-specific-inefficieny tracing is enabled.

Table 10.7 Fields in the TCBCONTROLB register

31	30 28	27 26	25 21	20	19	18	17	16	15	14	13 12	11	10 8	7	6 3	2	1	0
WE	0	TWSrcWidth	REG	WR	0	TRPAD	FDT	RM	TR	BF	TM	TLSIF	CR	Cal	TWSrcVal	CA	OfC	EN

In TCBCONTROLB:

TWSrcWidth: "0 1", which indicates that a 2-bit "source" field is included in the trace word to identify the VPE running the instruction, just as a multiCPU system would identify the CPU.

TWSrcWidth: "1 0", which indicates that a 4-bit "source" field is included in the trace word.

TWSrcVal: becomes writable, so the probe can set this value to a distinguishable one for each VPE.

FDT: set to 1 to indicate that Filtered Data Trace is enabled

TRPAD: set to 0 to enable software to access on-chip trace memory via the drseg mapped TCB Registers.

Table 10.8 Fields in the TCBCONTROLC register

30 30	29 28	27 23	22	21 15	14	13	12 9	8 5	4 2	1	0
Res	NumDO	Mode		Res			Res				

TCBCONTROLC contains new fields for multi-threading trace support, as described in [PDTRACETCB].

MTtraceTC: can be set to 1 to include the TC ID in trace data. Powers up as zero.

Figure 10.16 Fields in the TCBCONTROLE register

31 9	8	7 6	5	4	3	2	1	0
0	TrIDLE	0	PeCOvf	PeCFCR	PeCBP	PeCSync	PECE	PEC

Aside from TrIDLE the rest of the bits in TCBCONTROLE are enable and control bits for performance counter tracing

TrIDLE : is set by the hardware to indicate that the trace unit is not processing any data. This is especially useful when switching control from hardware to software and vice-versa. After turning trace off (recommended to turn *TraceControll[ON]* , *TCBCONTROLA[ON]* , and *TCBCONTROLB[EN]* off), this bit should be queried and if the trace unit is idle, then it is safe to change the trace control settings. After changing the settings, trace can be turned back on, and tracing resumes cleanly with the new control.

Figure 10.17 Fields in the TCBCONFIG register

31	30 25	24 21	20 17	16 14	13 11	10 9	8 6	5	4	3 0
CF1	0	TRIG	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

In TCBCONFIG:

CF1: read-only, reads zero because there are no more TCB configuration registers.

PiN: read-only, reads zero because the 1004K CPU is a single-issue (single pipeline) processor.

REV: reads 1, denoting compliance with revision 4.0 of the TCB specification.

REV: reads 3, denoting compliance with revision 6.0 of the TCB specification.

10.2.2 CP0 registers for the PDtrace™ logic

There are three categories of registers:

- *TraceControl* , *TraceControl2* and *TraceControl3 (*Figure 10.18/Figure 10.19 *)*: allow the software to take charge of what is being traced.

- *UserTraceData1* and *UserTraceData2* (Section 10.2.4 “UserTraceData1 reg and UserTraceData2 reg”): allows software to send a “user format” trace record, which can be interpreted by suitable trace analysis software to build interesting facilities.

- *TraceBPC* (Figure 10.21): controls whether and how individual EJTAG breakpoint trace triggers take effect.

Figure 10.18 Fields in the TraceControl Register

31	30	29	28	27	26	25	24	23	22	21	20 13	12 5	4	3	2	1	0
TS	UT	0	Ineff	TB	IO	D	E	K	S	U	ASID_M	ASID	G	TFCR	TLSM	TIM	On
0																	0

Figure 10.19 Fields in the TraceControl2 Register

31 30	29 12	11 7	6 5	4	3	2 0
SyPExt	0	Mode	ValidModes	TBI	TBU	SyP

Figure 10.20 Fields in the TraceControl3 register

31 14	13	12	11	10	9	8	7 3	2	1	0
0	PeCOvf	PeCFCR	PeCBP	PeCSync	PECE	PEC	0	TrIDLE	TRPAD	FDT

TS: set 1 to put software (manipulating this register) in control of tracing. Zero from reset.

UT: software can output a "user triggered record" (just write any 32-bit value to the *UserTraceData* register). There have been two types of user-triggered record, and this bit says which to output: 0 → Type 1 record, 1 → Type 2. This bit is deprecated as there are now two registers *UserTraceData1* and *UserTraceData2.* If a write to *UserTraceData1 or UserTraceData2* occurs, then the type is UT1 or UT2 respectively

Ineff: set to 1 to indicate that CPU-specific-inefficieny tracing is enabled.

TB: "trace all branch" - when 1, output all branch addresses in full. Normally, predictable branches need not be sent.

IO: "inhibit overflow" - slow the CPU rather than lose trace data because you can’t capture it fast enough.

D, E, K, S, U: do trace in various CPU modes: separate bits independently filter for debug, exception, kernel, supervisor and user mode. Set 1 to trace.

ASID_M, ASID, G: controls ability to trace for just one (or some) processes, recognized by their current ASID value as found in *EntryHi[ASID]*. Set the *G* ("global") to trace instructions from all and any ASIDs. Otherwise set *TraceControl[ASID]* to the value you want to trace and *ASID_M* to all 1s (you can also use *ASID_M* as a bit mask to select several ASID values at once).

TFCR: switch on to generate full PC addresses for all function call and return instructions.

TLSM: switch on to trace all D-cache misses (potentially including the miss address).

TIM: switch on to trace all I-cache misses.

On: master trace on/off switch - set 0 to do no tracing at all.

The read-only fields in *TraceControl2* provide information about the capabilities of your PDtrace system. That system may include a plug-in probe, and in that case the *TraceControl2[SyP]* field may read as garbage until the probe is plugged in.

The first four fields are for tracing code running on MT CPUs:

CPUIdV, CPUId: when *CPUIdV* is set, trace data will only be generated by code run by the VPE identified in *CPUId*. Ignored if *TCV* is set.

TCV, TCNum: when *TCV* is set, trace only instructions run by the TC whose number is stored in t*TCNum*.

Mode: whenever trace is turned on, you capture an instruction trace. *Mode* is a bit mask which determines what load/store tracing will be done[39]. It's coded like this:

Bit No Set	*What gets traced*
`0`	PC
`1`	Load addresses
`2`	Store addresses
`3`	Load data
`4`	Store data

However, see *TraceControl2[ValidModes]* (description below) for what your PDtrace unit is actually capable of doing. Bad things can happen if you request a trace mode which isn't available.

TraceControl2[ValidModes]: what is this PDtrace unit capable of tracing?

ValidModes	*What can we trace?*
`00`	PC trace only
`01`	Can trace load/store addresses
`10`	Can trace load/store addresses and data

TraceControl2[TBI,TBU]: best considered together, these read-only bits tell you whether there is an on-chip trace memory, on-probe trace memory, or both - and which is currently in use.

TBI	*TBU*	*On-chip or probe trace memory?*
`0`	`0`	only on-chip memory available
`0`	`1`	only probe memory available
`1`	`0`	Both available, currently using on-chip
`1`	`1`	Both available, currently using probe

TraceControl2[SyP]: read-only field which lets you know how often the trace unit sends a complete PC address for synchronization purposes, counted in CPU pipeline clock cycles. The period is $2^{(SyP + 5)}$. Valid periods are 2^{5} to 2^{12} .

TraceControl2[SyPExt]: This is an extension to the SyP. It is useful when a higher number of cycles is desired between synchronization events. The same formula applies as that described above, except that it applies to the juxtaposition of SyPExt and SyP. The period is $2^{(SyPExt ,SyP + 5)}$. Valid periods are 2^{5} to 2^{31}. If the user tries to specify a period above 2^{31}, the behavior is unpredictable.

TraceControl3[FDT]: set to 1 to indicate that Filtered Data Trace is enabled

TraceControl3[TRPAD]: read-only bit that is loaded from *TCBControlB*$_{\text{TRPAD}}$.

TraceControl3[TrIDLE] :read-only bit that is set by the hardware to indicate that the trace unit is not processing any data. This is especially useful when switching control from hardware to software and vice-versa. After turning trace off (recommended to turn *TraceControll[ON]* , *TCBCONTROLA[ON]* , and *TCBCONTROLB[EN]* off) , this bit should be queried and if the trace unit is idle, then it is safe to change the trace control settings. After changing the settings, trace can be turned back on, and tracing resumes cleanly with the new control.

The rest of the bits in *TraceControl3* enable and control performance counter tracing.

[39] Prior to v4 of the PDtrace specification, this field was in *TraceControl*, and was too small to allow all conditions to be specified independently.

10.2.3 JTAG triggers and local control through TraceIBPC/TraceDBPC

Recent revisions of the PDtrace specification have defined much finer controls on tracing. In particular, you can now trace only cycles matching some “breakpoint” criteria, and there is a two-stage process where cycles are traced only after an “arm” condition is detected. The new fields are shown in Figure 10.21

Figure 10.21 Fields in the TraceIBPC/TraceDBPC registers

	31 30	29	28	27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
TraceIBPC	0	PCT	IE	ATE						IBPC3	IBPC2	IBPC1	IBPC0
TraceDBPC			DE									DBPC1	DBPC0

In either *TraceIBPC* or *TraceDBPC*:

PCT: set to 1 and a performance counter trigger signal is generated when an EJTAG breakpoint match occurs.

IE,DE: master 1-to-enable bit for triggers from EJTAG instruction and data breakpoints respectively.

ATE: Read-only bit which lets you know whether the additional trigger controls such as ARM, DISARM, and data-qualified tracing (introduced in v4.00 of the PDtrace specification) are available - which they may be on the 1004K CPU. This bit is deprecated and reads as zero.

IBPC8-0, DBPC8-0: each three-bit field encodes tracing options independently, for up to nine EJTAG I- and D-side breakpoints (this is generous: your 1004K CPU will typically have no more than 4 I- and 2 D-breakpoints).

Each entry can be set as follows:

xBPC field	*Description*
0	Stop tracing (no effect if off already).
1	Start tracing (no effect if on already).
2	Trace instructions which cause this trigger.

However, do *TraceIBPC*/*TraceDBPC* exist in your system? They will be there only if you have an EJTAG unit (does *Config1[EP]* read 1?), and that unit has at least one breakpoint register - check that at least one of *DCR[DB,IB]* is set (as described in).

10.2.4 UserTraceData1 reg and UserTraceData2 reg

Write any 32-bit value you like here and the trace unit will send a “user” record (if only one *UserTraceData* register exists, then there are two “types” of user record, and which you output depends on *TraceControl[UT]*, see above). However if two *UserTraceData* registers exist then writing to *UserTraceData1* will generate a trace record with type UT1, and writing to *UserTraceData2* will generate a trace record with type UT2. You need to send something your trace analysis system will understand, of course! Perhaps it’s worth noting that this “user” is local debug software, and doesn’t mean low-privilege software running in “user mode” - which of course would not be able to access this register. CP0 access rules apply when writing to this “user” register.

10.2.5 Summary of when trace happens

The many different enable bits which control trace add up to (or strictly “and” up to) a whole bunch of reasons why you won’t get any trace output. So it may be worth summarizing them here. So:

- If software is in charge (that is, if *TraceControl[TS]*`==1`) then:
  - *TraceControl[On]* must be set.
  - At least one of the CPU mode filter bits *TraceControl[D,E,S,K,U]* must be set 1 to trace instructions in debug, exception, supervisor, kernel or user-mode respectively. Mostly likely either just *TraceControl[U]* will be set (to follow just one process in a protected OS), or *TraceControl[E,S,K,U]* to follow all the software at bare-iron level (but not to trace EJTAG debug activity);
  - Either *TraceControl[G]* is set (to trace everything regardless of current ASID) or *TraceControl[ASID]* (as masked by *TraceControl[ASID_M]*) matches the current value of the CPU-under-test's *EntryHi[ASID]* field.
  - The signal *PDI_TraceOn* is asserted by the trace block. This will typically be true whenever the probe is plugged in and connected to software.
  - As above there are *D,E,S,K,U,G* and *ASID* bits (there isn't an "ASID_M" in this case) which must be set appropriately in the JTAG-accessible *TCBCONTROLA* register, which is not otherwise described here.

Whether JTAG or *TraceControl* is in charge, then:

- There must have been a cycle recently when there was an "on trigger", that is:
  - The CPU tripped an EJTAG breakpoint with the *IBCn[TE]*/*DBCn[TE]* bit set to request a trace trigger (for I-side and D-side respectively);
  - *TraceIBPC[IE]*/*TraceDBPC[DE]* (respectively) was set to enable triggers from EJTAG breakpoints;
  - the appropriate *TraceBPC[IBPCx]*/*TraceBPC[DBPCx]* field has some kind of "on" trigger - and if this trigger is conditional on "arm" there must have been an arm event since system reset or any disarm event; or the trigger unconditionally turns trace on.
- And since the on-trigger time, there must not have been a cycle which acted as an "off trigger", that is:
  - The CPU tripped an EJTAG breakpoint with the *IBCn[TE]*/*DBCn[TE]* bit set, and *TraceBPC[IE]*/*TraceBPC[DE]* (respectively) were still set;
  - where the appropriate *TraceIBPC[IBPCn]*/*TraceDBPC[DBPCn]* fields is set to disable triggering (subject to arming).

If there is more than one breakpoint match in the same cycle, an "on" trigger wins out over any number of "off".

10.3 CP0 Watchpoints

Some cores may be built with no EJTAG debug unit to save space, and some debug software may not know how to use EJTAG resources. So it may be worth configuring the four non-EJTAG CP0 watchpoint registers. In 1004K cores you get two I-side and two D-side registers (unless, of course, the core was built without them - check *Config1[WR]*).

These registers provide the interface to a debug facility that causes an exception if an instruction or data access matches the address specified in the registers. Watch exceptions are not taken if the CPU is already in exception mode (that is if *Status[EXL]* or *Status[ERL]* is already set).

Watch events which trigger in exception mode are remembered, and result in a "deferred" exception, taken as soon as the CPU leaves exception mode.

This CP0 watchpoint system is independent of the EJTAG debug system (which provides more sophisticated hardware breakpoints).

The *WatchLo0-3* registers hold the address to match, while *WatchHi0-3* hold a bundle of control fields.

10.3.1 The WatchLo0-3 registers

Used in conjunction with *WatchHi0-3* respectively, each of these registers carries the virtual address and what-to-match fields for a CP0 watchpoint.

Figure 10.22 Fields in the WatchLo0-3 Register

31 3	2	1	0
VAddr	I	R	W
	0	0	0

WatchLo0-3[VAddr]: the address to match on, with a resolution of a doubleword.

WatchLo0-3[I,R,W]: accesses to match: I-fetches, Reads (loads), Writes (stores). 1004K cores have separate I- and D-side watchpoints, so you'll find that the I-side *WatchLo0-1[R]* and *WatchLo0-1[W]* is fixed to zero, while for the D-side-only watchpoint, *WatchLo2-3[I]* will be zero.

10.3.2 The WatchHi0-3 registers

Figure 10.23 Fields in the WatchHi0-3 Register

31	30	29 24	23 16	15 12	11 3	2	1	0
M	G	0	ASID	0	Mask	I	R	W
X								

WatchHi0-3[M]: the *WatchHi0-3[M]* bit is set whenever there is one more watchpoint register pair to find; your software should use it (starting with *WatchHi0*) to figure out how many watchpoints there are. That's more robust than reading the CPU manual...

WatchHi0-3[G,ASID]: *WatchHi0-3[ASID]* matches addresses from a particular address space (the "ASID" is like that in TLB entries) — except that you can set *WatchHi0-3[G]* ("global") to match the address in any address space.

WatchHi0-3[Mask]: implements address ranges. Set bits in *WatchHi0-3[Mask]* to mark corresponding *WatchLo0-3[VAddr]* address bits to be ignored when deciding whether this is a match.

WatchHi0-3[I,R,W]: read your *WatchHi0-3* after a watch exception, and these fields tell you what type of access (if anything) matched.

Write a 1 to any of these bits in order to *clear* it (and therefore prevent the exception from immediately happening again). This behavior is unusual among CP0 registers, but it is quite convenient: to clear a watchpoint of all the exception causes you've seen just read the value of *WatchHi0-3* and write it back again.

10.4 Performance counters

Performance counters are provided to allow software to monitor the occurrence of events of interest within the core, and can be very useful in analyzing system performance.

1004K family CPUs are fitted with counters, each of which can be set up to count one of a large choice of different events. Each 32-bit counter is accompanied by a control register whose layout is shown in Figure 10.24.

Two different configurations are found on 1004K cores, distinguished by the *Config7[PCT]* bit (see notes to Figure C-3:

- If *Config7[PCT]* is zero there are four "global" counters: unlike almost all the other CP0 registers, the performance counters are *not* replicated per-VPE: the CPU has four counters, which either VPE may use;
- If *Config7[PCT]* reads 1, there are two counters, but replicated per TC. Note that just because a particular control/count pair is only directly accessible by software running on its own TC, all the counters can count whatever events for whatever TC and VPE you choose.

There's no direct way of seeing which organization your CPU is using. But you can find out how many registers the software can see by inspecting the *PerfCtl[M]* bits (see below); to check whether the counters are per-TC, check whether a change made to a register is reflected in the value returned from the viewpoint of another TC. The **`mftr`** instruction is your friend here — see Section 2.8 "MIPS® Multithreading ASE - new instructions".

1004K is a multi-threading CPU, and you can optionally count only events associated with a particular thread (by its TC number), or even those events associated with threads affiliated to a particular VPE. After some thought, I haven't documented in detail when you might get a different count if you narrow to a particular VPE or TC. In most cases it's obvious whether it makes sense to count a particular event for just one TC or VPE: where it's not obvious, experiment.

Figure 10.24 Fields in the PerfCtl Registers

31	30	29 22	21 20	19 16	15	14 12	11 5	4	3	2	1	0
M	0	TCID	MT_EN	VPEID	PCTD	0	Event	IE	U	S	K	EXL
X								0				

Software should not assume knowledge of how many counters there are. Instead, check using the *PerfCtl[M]* bit (which indicates "at least one more").

Then the fields are:

M: Reads 1 if there is another *PerfCtl* register after this one.

TCID: the TC number of the thread whose events should be counted, if just-one-TC counting is enabled (i.e. *MT_EN*==10 binary.)

MT_EN: available to restrict counting to events which are attributable to a particular VPE or TC:

MT_EN *value*	*What gets counted?*
00	Events from all TCs & VPEs (i.e., don't filter)
01	Count events from all TCs affiliated to the VPE specified in the *VPEID* field. Some events can't be tied to a particular VPE - use common sense.
10	Count events only for the TC specified by the *TCID* field. Again, some events are not TC-specific.
11	Reserved

VPEID: defines the VPE all of whose TC's events should be counted, if just-this-VPE counting is enabled (i.e. *MT_EN*==01 binary.)

PCTD: controls whether this performance counter will be included in the performance counter trace mode of PDtrace. Setting the bit will prevent the tracing of this counter

Event: determines which event this counter will count; see Table 10.9 below. Note that the odd-numbered and even-numbered counters mostly count different events, though some particularly important events can use any of the counters.

IE: set to cause an interrupt when the counter "overflows" into its bit 31. This can either be used to implement an extended count, or (by presetting the counter appropriately) to notify software after a certain number of events have happened.   The interrupt is implemented by taking a set of signals (usually *SI_PCI nt*- one per VPE) out of the core, which the system integrator will have sent back in, each as one of the core's interrupt inputs. The output signal activated will depend on the VPE affiliation of the thread which last wrote to the control register, which will normally be what you want.

U, S, K, EXL: count events in User mode, Supervisor mode, Kernel mode and Exception mode (i.e. when *Status[EXL]* is set) respectively. Set multiple bits to count in all cases.

The events which can be counted in the 1004K core are in Table 10.9. Blank fields are reserved. But before you get there, take a look at the next sub-section...

10.4.1 Reading the event table.

There are a lot of events you can count. It's relatively cheap to wire another signal from the internals of the core into a counter. It's time consuming and expensive to formulate a signal which represents exactly what a software engineer might want to count, and even more expensive to test it. Where the definitions in Table 10.9 are clear and simple, they're usually exactly right. Where they seem more obscure, tread carefully, and don't just blame the author of this manual (though sometimes it is my fault!) When you use a counter, use it first on a piece of code where you know the answer, and check you're really counting what you think you are.

When reading the table:

- *T, V, P*: relevant only to a multithreading CPU. In the "Type" column, mark an event which can be filtered per-TC, per-VPE or is just global (respectively). Per-TC events can be counted per-VPE, and per-VPE events can be counted globally. When you count per-TC events per-VPE or globally the counter will advance in any cycle where the event happens for any TC under consideration. Counters never advance faster than once per clock.

- *IFU*: is the “instruction fetch unit” of the CPU pipeline. We can’t describe some events without referring to the inside of the CPU. You might like to look back at Section 3.1 “The 1004K™ core pipeline and multithreading”.

- *Replay*: when an instruction will block for a long period, sequentially-later instructions from the same TC which have got into the main pipeline must be discarded. These instructions will usually have been retained in the “skid buffer” of the IFU, so the IFU queues can be adjusted so that when the instruction unblocks, the TC can continue correctly from the following instruction. This sequence is called a “replay” and these events count the pipeline bubbles which result.

- *Refetch*: if you'd like to do a replay but the relevant instructions are not available in the skid buffer, the IFU must be instructed to discard all stored instructions for the TC and fetch them again. This event counts the number of pipeline bubbles which result.

- *Stall*: in general, “stall” counters count the cycles when the whole pipeline is blocked and no TC can make forward progress. If this type of counter is set for a particular TC, it will only count if this TC is causing the stall.

  But subunits causing a stall can also signal a “long stall”, and the main pipeline takes that as a cue to deschedule the blocked TC until the condition is resolved. The counters documented as “stall” or “stalled” do not count time while one TC is blocked but others continue to run.

- *Blocked cycle*: “events” like this count all and any cycles when a TC is blocked by something.

- *LDQ, FSB, WBB*: CPU queues, described in Section 5.3.1 “Read/write ordering and cache/memory data queues in the 1004K‘ CPU”.

- *Instruction fetch events*: these include I-cache, ITLB and JTLB events. They are not as directly related to the instructions in your program as you might think:

  - 1004K family CPUs have a 64-bit wide interface to the I-cache and fetch two instructions at once.

  - After a cache miss is resolved, the IFU re-fetches the missed data; the counters will count this twice.

  - The IFU always reads instructions ahead, and on a branch or exception some of the instructions fetched will never be executed. Moreover, the IFU's branch predictors sometimes cause it to fetch speculatively from a predicted branch target which turns out to be wrong: those speculative instructions will never be executed either.

  - If there's an exception-causing address error during I-fetch, it won't be counted.

- *Single-threaded mode (ST mode)*: when only one TC is eligible for scheduling, the 1004K core hardware is in “ST mode”. In ST mode some blocking events which would otherwise have been dealt with by suspending the thread and possibly replaying an instruction are handled by a whole-pipeline stall instead. This saves power, and is more efficient in the case where no other TC becomes runnable until the stall condition is resolved.

- *Exceptions in a branch delay slot*: are handled by internally setting the exception-return register *EPC* to point to the branch instruction. After the exception is handled and control returns, the branch instruction is re-executed: all MIPS branch instructions are contrived so the re-execution does exactly the same thing as the first time. But the instruction is “really” run twice, and any performance count will show that.

Table 10.9 Performance counter events

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
0	Cycles	P	Cycles	P
1	Instructions completed	T	Instructions Completed	T
2	Branch instructions completed.	T	Branch mispredictions	T
3	`jr $31` (return) instructions	T	`jr $31` predicted but guessed wrong	T
4	`jr` (not `$31`) instructions	T	`jr $31` not predicted (the return predictor only works for one TC at a time).	T
5	ITLB accesses. There will be one for every I-fetch in a translated address region.	T	ITLB misses. Note that if two TCs cause "the same" ITLB miss in quick succession, that will only be counted once.	T
6	DTLB accesses.	T	DTLB misses	T
7	JTLB instruction accesses (same as ITLB misses).	T	JTLB I-misses: this counts TLB misses *and* TLB invalid conditions on I-fetch.	T
8	JTLB data accesses (same as DTLB misses)	T	JTLB D-miss: counts TLB misses + TLB invalid on D-access.	T
9	Instruction cache accesses. Since pairs of instructions are fetched over the 64-bit bus, this is only very approximately one per two instructions. And that's *every* access: even though the instruction ends up dropped because of an exception, or a thread becoming blocked. And instructions which are refetched will end up being counted twice.	T	Instruction cache misses. Includes misses resulting from fetch-ahead and speculation.	T
10	Data cache load/stores	T	D-cache writebacks (strictly, the number of D-misses or cacheops which trigger a writeback.)	T
11	Loads/stores which miss in D-cache	T	Loads/stores which miss in D-cache	T
12	reserved	-	reserved	-
13	Store misses	T	Load Misses	T
14	Integer instructions completed	T	FPU instructions completed (not including loads and stores)	T
15	Loads completed (including FP loads)	T	Stores completed (includes FP stores)	T
16	`j`/`jal` instructions completed	T	MIPS16 instructions completed	T
17	no-ops completed. Early revision cores count only strict `nop` instructions, but later ones count any 3-operand instruction which discards its output by writing register `$0`.	T	Integer multiply/divide unit instructions completed	T
18	Cycles where the main pipeline (RF stage) does not advance. This is either because there is no instruction scheduled, or because the ALU is backed up and can't accept an instruction	P	Refetches: that is, events where IFU is made to re-issue instructions which were already scheduled once.	T
19	`sc` instructions completed	T	`sc` instructions failed	T
20	Prefetch instructions to cached addresses	T	Prefetch instructions completed with cache hit	T
21	L2 cache writebacks	P	L2 cache accesses	P
22	L2 cache misses	P	Single-bit errors corrected in L2	P
23	Exceptions taken	T	Cycles spent in "Single Threaded Mode".	T

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
24	Cycles when main pipeline is stalled while the LSU has to do a "replay". A good example of a replay is when the fill buffer gets full and needs to be emptied out to make forward progress. To empty out the buffer, the LSU has to take control of the cache which is currently being accessed by other in-flight LSU instructions. To accomplish this, the pipeline is stalled, the FSB accesses the cache to empty out its data, and then the instructions that were in flight are "replayed" to get their data from the cache.	T	"Refetches": Counts all replayed instructions (instructions which are send back to IFU to be refetched and reissued)). If an instruction has been replayed multiple times, you get a count for each event.	T
25	Cycles when no instructions are available to issue for any TC	P	Cycles when main pipeline stops because an ALU operation is taking more than one clock	P
26	DSP Instructions Completed	T	ALU-DSP Saturations Done	T
27			MDU-DSP Saturations Done	T
28-31	*Available to count implementation-specific events signalled by wires from configurable interfaces.*			
28	Available for customer PM event	T	Available for customer CP2 event	T
29	Available for customer ISPRAM event	T	Available for customer DSPRAM event	
30	Available for CorExtend event	T		
31	Available for external yield manager event.	T	Custom ITC event	T
32	ITC Loads. If a TC is halted or takes an exception, a pending ITC operation will be aborted, then later retried. Each retry is counted.	T	ITC Stores issued. Invisible retries counted too, as for loads.	T
33	Uncached Loads	T	Uncached Stores	T
34	**`fork`** Instructions completed	T	**`yield`** instructions completed	T
35	CP2 register-to-register instructions completed	T	**`mfc2`**/**`mtc2`** instructions completed	T
36	Intervention stalling main pipeline	P	Intervention response stalled on miss	P
37-46	Count number of cycles (most often "stall cycles", i.e. time lost), not just number of events. See note on stall cycles above.			
37	I-cache miss blocked cycles - counts cycles when the TC has no instruction to issue following an I-fetch miss. This ignores the stalls due to ITLB misses as well as the 4 cycles following a redirect.	T	D-cache miss blocked cycles - counts cycles when TC is blocked when an instruction uses a register value which is subject to a load miss.	T
38	SYNC Stall Cycles	T	FSB Index Conflict Stalls	P
39	D-miss cycles	P	L2 miss cycles	P
40	Uncached access block cycles	T	ITC stall cycles: when no instruction for any TC can be issued, and a TC selected for counting is waiting for an ITC operation	T
41	MDU stall cycles - note that it's possible for the MDU to indicate a "long stall" where the TC waiting for the MDU gets suspended - that wait will *not* be counted here.	T	FPU stall cycles	T
42	CP2 stall cycles	T	CorExtend stall cycles	T

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
43	ISPRAM stall cycles - when no instruction can be issued because the IFU has run out of instructions, after the ISPRAM sent a “not ready” indication (which requires a retry). Doesn't include a count for the 4 cycles after a redirect.	T	DSPRAM stall cycles	T
44	CACHE instruction stall cycles	P	Cycles spent stalled waiting for something which we’d normally deal with by “parking” the TC, but where we’ve stalled because we’re in ST mode (ie no other TC has instructions to schedule). Long stalls in this sense include those which result from cache misses or waiting for a divide or square-root instruction.	P
45	Load to Use stalls	T	Stalls when a load/store base register was computed by the preceding instruction.	T
46	Read-CP0-value interlock stalls.	T		
47	Relax bubbles	V		
48	IFU FB full refetches: count up when the IFU has to refetch an address because the FB was full on a miss.	T	FB entry allocated	P
49	EJTAG Instruction triggers	T	EJTAG data triggers	T
50-55	Monitor the state of various FIFO queues relating to loads and stores, as described in Section 5.3.1 “Read/write ordering and cache/memory data queues in the 1004K‘ CPU”.			
50	FSB < 1/4 full	P	FSB 1/4-1/2 full	P
51	FSB > 1/2 full	P	FSB full pipeline stalls	P
52	LDQ < 1/4 full	P	LDQ 1/4-1/2 full	P
53	LDQ > 1/2 full	P	LDQ full pipeline stalls	P
54	WBB < 1/4 full	P	WBB 1/4-1/2 full	P
55	WBB > 1/2 full	P	Cycles when whole CPU is stopped because an instruction needs to write data out of the core, but all write buffer entries are full.	P
56	Intervention Hits	P	All Interventions	P
57	All Invalidates	P	Invalidate Hits	P
58	Evictions	P	Writeback	P
59	ST_Inval	P	ST_Exclusive	P
60	ST Store to S	P	ST Downgrade	P
61	Request Latency to Self Intervention	P	Request Count for SI Latency	P
62	Request Count for Resp. Latency	P	Request Latency to Read Response	P
63	Reserved			
64	SI_PCEvent[0] - System specific event 0	P	SI_PCEvent[1] - System specific event 1	P
65	SI_PCEvent[2] - System specific event 2	P	SI_PCEvent[3] - System specific event 3	P
66	SI_PCEvent[4] - System specific event 4	P	SI_PCEvent[5] - System specific event 5	P
67	SI_PCEvent[6] - System specific event 6	P	SI_PCEvent[7] - System specific event 7	P
68-127	Reserved			

Appendix A

References

1004K™ core family manuals

[SUM]:“MIPS32® 1004K® Processor Core Family Software User’s Manual”, MIPS Technologies document MD00622.

[ERRATA]:“MIPS32® 1004K® Processor Core Family RTL Errata Sheet”, MIPS Technologies document MD00600. Available only to core licensees or by arrangement.

[INTGUIDE]:“MIPS32® 1004K® Processor Core Family Integrator's Guide”, MIPS Technologies document MD00620, available to core licensees, describes the options available with the core.

[CPS_USM]:“MIPS32® 1004K® Coherent Processing System User’s Manual”, MIPS Technologies document MD00597, available to core licensees, describes the other blocks within the CPS.

[34KC_DATA]:“MIPS32 34Kc™ Datasheet”, MIPS Technologies document MD00418.

[34KF_DATA]:“MIPS32 34Kf™ Datasheet”, MIPS Technologies document MD00419: these are the basic hardware introductions, for family members without and with floating point unit.

[74KC_DATA]:“MIPS32 74Kc™ Datasheet”, MIPS Technologies document MD00496.

[74KC_DATA]:“MIPS32 74Kc™ Datasheet”, MIPS Technologies document MD00496.

Other Manuals from MIPS Technologies

First of all, basic architectural documents.

[MIPS32]: The MIPS32 architecture definition series in three volumes:

[MIPS32V1]: “Introduction to the MIPS32 Architecture”, MIPS Technologies document MD00080.

[MIPS32V2]: “The MIPS32 Instruction Set”, MIPS Technologies document MD00084.

[MIPS32V3]: “The MIPS32 Privileged Resource Architecture”, MIPS Technologies document MD00088.

Although cores in the 1004K family are 32-bit cores, the optional floating-point unit is a 64-bit one, and is as described in:

[MIPS64V2]:“The MIPS64 Instruction Set”, MIPS Technologies document MD00085.

Then there are some architectural extensions:

[MIPSMT]:“The MIPS MT Application-Specific Extension to the MIPS32 Architecture”, MIPS Technologies document MD00376.

[MIPSDSP]:“The MIPS DSP Application-Specific Extension to the MIPS32 Architecture”, MIPS Technologies document MD00372.

[DSPWP]: “Effective Programming of the 24KE and 34K family Cores for DSP Code”, MIPS Technologies white paper, document number MD00475.

[MIPS16e]:“The MIPS16e™ Application-Specific Extension to the MIPS32 Architecture”, MIPS Technologies document MD00074.

[CorExtend]:“How To Use CorExtend® User-Defined Instructions”, MIPS Technologies document MD00333.

[CorExtend-24K]:“CorExtend Instructions Integrator’s Guide for MIPS32 24K, 24KE and 34K Pro Series™ cores”, MIPS Technologies document MD00348.

[EJTAG]:“MIPS® EJTAG Specification”, MIPS Technologies document MD00047.

[PDTRACEIF]:“PDtrace™ Interface Specification”, MIPS Technologies document MD00136.

[PDTRACEUSAGE]:“PDtrace™ and TCB Usage Guidelines”, MIPS Technologies document MD00365.

[PDTRACETCB]:“MIPS® PDtrace™ Interface and Trace Control Block Specification”, MIPS Technologies document MD00439. Current revision is 4.30: you need revision 4 or greater to get multithreading trace information.

[L2CACHE]:“MIPS® SOC-it® L2 Cache Controller Users Manual”, MIPS Technologies document MD00525.

Books about programming the MIPS® architecture

[SEEMIPSRUN]: “See MIPS Run, 2nd Edition”, author Dominic Sweetman, Morgan Kaufmann ISBN 1-55860-410-3. A general and wide-ranging programmers introduction to the MIPS architecture, updated in 2006 to reflect the current version of [MIPS32].

[MIPSPROG]:“MIPS Programmers Handbook”, Erin Farquar & Philip Bunce, Morgan Kaufmann ISBN 1-55860-297-6. Restricted to the MIPS I instruction set but with a lot of assembler examples.

Other references

[IEEE754]:“IEEE Standard 754 for Binary Floating-Point Arithmetic”, published by the IEEE, widely available on the web. Surprisingly comprehensible.

C language header files

Header files are available as part of the free-for-download “SDE Lite” subset available from MIPS Technologies’ website. You’ll find them under*.../sde/include/mips/*. In particular:

[m32c0 h]:C definitions referred to in this manual for the names and fields of standard MIPS32 CP0 registers.

[m32tlb.h]:C definitions and constants associated with the basic address space and TLB programming.

[mt h]:C definitions for CP0 registers and other programmable resources of the MIPS MT system.

Appendix B

Glossary

ASE: “Application-Specific Extension” to an instruction set. The acronym is used by MIPS Technologies to describe optional add-ons to the core MIPS32/MIPS64 architecture. The multi-threading package is the “MIPS MT ASE” and there’s a bunch of others including the recent “DSP ASE” which adds computational instructions relevant to media-stream signal processing.

Co-processor:the MIPS architecture reserves some parts of the instruction set for “co-processors” - which have a few standard instructions, some instruction encoding space and standard registers. Co-processors can be standard but optional (like the floating point unit); a space for customers to build their own logic (like CP2); or, in the case of “co-processor zero”, just a way to separate the encodings of critical (and certainly not optional) processor control operations and registers.

Co-processor zero:see CP0 below.

CP0: MIPS computers use a bunch of register fields for most CPU control purposes. They’re accessible only in high-privilege mode, since they’re part of the protection system for a protected OS. The registers and the instructions used to access them are defined using a built-in instruction set extension mechanism which conceives of four sets of instruction encodings reserved for “co-processors”: the control register set, which must be present in any MIPS32 CPU, are “co-processor zero”.

CP0 hazard:a hazard which makes some instruction sequences involving privileged operations (and particularly privileged “CP0” registers) illegal. Until quite recently OS programmers were expected to deal with CP0 hazards by inserting “enough” **`nop`** instructions between producers and consumers of CP0 values and state; but with Revision 2 of [MIPS32] there are better ways described in Section 7.1, "Hazard barrier instructions".

Dispatch Scheduler:the logical block of a MIPS MT multithreading CPU which determines which thread to favor when issuing instructions into the sequential main pipeline.

EMT: (“*Explicit Multithreading****”)*** software which is deliberately written in terms of closely coupled (i.e. memory sharing) concurrent threads. and therefore can directly benefit from multi-threading features of the underlying CPU.

Gating Storage:a kind of special uncacheable memory recognized by a multithreading CPU. It’s suitable for use for accessing locations where the load/store will not be completed until some event external to the thread, with no obvious maximum waiting time.

From the software’s point of view, gating storage is synchronous: no instruction, side-effect or exception from the after the gating load/store is permitted unless and until the load/store completes. A load/store to gating storage may be aborted at any time before it completes, and this will be signalled as a precise exception whose return address is the load/store instruction[40].

From the hardware’s point of view, gating storage has a special interface to the core. The storage subsystem must signal a completed store, and the core can (at any time while waiting for a load/store to complete) ask the storage subsystem to abort the operation. An aborted operation must be “as if it never happened”.

There will be some handshaking between the core and the storage subsystem to avoid a race condition between completion and abort. In some circumstances, software trying to abort a gated load/store will fail, and will be told

[40] Or the branch instruction in whose delay slot the load/store lives - usual MIPS exception rules.

that the operation completed before it could be aborted, and will then have to cope with whatever side-effects the operation had.

Hazard:(or “pipeline hazard”) - an architectural requirement which requires you to avoid some instruction sequences. Historical MIPS CPUs had some interesting hazards (like the “load delay slot” and an exception corner case on multiply operations). For a long time MIPS CPUs have only had hazards on code sequences using privileged operations, see CP0 hazard.

Interrupt exempt: in a MIPS MT CPU like the 1004K core a TC may be marked as interrupt-exempt by setting *TCStatus[IXMT]*; then any interrupt presented to the VPE will never cause an exception to that TC. If all TCs belonging to a VPE are marked interrupt-exempt, that’s yet another way of disabling all interrupts.

Inter-Thread Communication storage:a generalized form of empty/full storage provided with the 1004K core, and attaching to the gating storage (see above) interface. It’s described in Section 3.3, "Inter-thread communication storage (ITC)".

ITC: short for “Inter-Thread Communication storage” as above.

ITC Cell:one location of ITC storage. A cell stores 32 bits of data, but has multiple views at different memory locations, each of which behaves differently.

*Pipeline hazard:*see *Hazard* above.

Redirect:what happens in the pipeline when the 1004K core encounters an unpredictable or wrongly-predicted branch instruction. The branch address and condition are finally available by the end of the “EX” pipeline stage (see Section 3.1, "The 1004K™ core pipeline"); at this point all instructions in the pipeline or fetch unit for this thread must be discarded, and instructions fetched from the now-correct instruction instead. That’s a redirect.

Relax:used for the extra “bogus TC” on the 1004K core which does nothing. The external thread scheduling “policy manager” (see below) has “relax” signals alongside those for real threads; when the “relax” condition has higher priority than any running threads the CPU does nothing for a cycle. This is a way of turning down the CPU (possibly saving energy) when no thread is urgent. See Section 3.2.3, "MIPS Policy managers included with the 1004K‘ core family".

Shadow register set:an extra set of general-purpose registers which can be automatically used in an interrupt handler (or other exception handler). Applications on MIPS32 architecture CPUs can use these shadow registers to reduce the overhead of interrupt handlers, both by retaining quickly-used state in the shadow registers and by avoiding the need to save and restore the state of the interrupted thread. See Section 7.4, "Shadow registers".

For software compatibility, the 1004K core can recycle one or more otherwise-unused TCs’ registers as a shadow set; see Section 4.3.6, "TCs recycled as Shadow registers".

Skid buffer:in a busy multi-threading CPU threads will block very frequently. When a thread blocks there may well be later instructions from the same thread in the pipeline: you can’t stop the pipeline without holding up all the other threads, and you can’t let this thread’s later instructions complete until this thread is unblocked. So those instructions must be discarded. It would be a problem if we had a full *Redirect* every time a thread blocked, so the 1004K core’s instruction fetch unit incorporates a “skid buffer” for each thread, which remembers the last couple of instructions issued. When a thread blocks and instructions are discarded from the main pipeline, the skid buffer can be backed up ready for the thread to be unblocked without having to fetch a whole lot more instructions.

TC: the logic and registers implementing a minimal thread state in the MIPS MT ASE (from “Thread Context”). A TC has at least its own PC, general-purpose registers and some other necessary bits and pieces. One or more TCs accessing the same complete set of CP0 registers make up a *VPE*.
The “Tera” project used the word “stream” for this.

Thread:a computation consisting of a set of computer instructions read and activated in their programmed order.

Operating systems often use the word “thread” specifically for application-software-visible explicit threads scheduled by the operating system kernel. But any code which is entered by something other than an application-programmed branch forms a separate thread by this definition: an interrupt handler, for example.

Thread context:the complete state of a computation as held within the CPU. The thread state excludes (1) data stored in memory, (2) state which is inaccessible to the instruction stream (such as CP0 register contents as seen by a user task) and (3) state which is insignificant (such as cache contents, which generally make no difference to the underlying memory image).

What comprises the thread state varies according to what sort of software is running. For a Linux OS interrupt handler thread on a conventional MIPS CPU the CP0 registers are part of the thread context, but for a Linux application thread they’re not. The thread state always (of course) includes the “program counter” (“PC”).

Policy Manager (PM):an implementation-dependent piece of logic (located outside of the MIPS core) which receives thread scheduling information from the CPU and hints from the *TCSchedule*/*VPESchedule* registers, and uses those and other customer-chosen inputs to propose a priority for the various TCs. The interface is designed to permit the policy manager to substantially define scheduling strategy, without the system being prone to failures caused by the inevitable delay between thread events and the PM’s response to them reaching the in-core thread scheduler.

Program Counter (PC):A software concept - the address of the next instruction that the thread will execute. It’s realization in hardware is somewhat elusive in a pipelined CPU implementing the MIPS architecture. However, it makes a comeback as a hardware-visible thing with the MIPS MT ASE; it is well-defined in hardware for any thread loaded into a TC but which is currently stopped (that is, there are no non-speculative instructions in flight). Such threads keep their PC in the *TCRestart* register.

Virtualizable:a CPU feature which can be allocated from a user-privilege program and (transparently to the user program) provided by either the hardware or automatic OS assistance.

So when an OS offers “virtual memory” there’s memory which is accessible by the user program - but when there isn’t enough memory the user program wanders off the ready-mapped pages, generates an exception which the OS can catch and map some more memory before restarting the application (back exactly where it was when it tried to reference the memory which wasn’t there).

MIPS MT resources - notably the TC which runs a concurrent thread - are defined to be virtualizable too. User programs can do their own thread creation and termination using the **`fork`**/**`yield $0`** instructions, with an OS intervening when no TC is available.

VSMP:a system with multiple concurrent threads running in separate VPEs (see the next entry), which behaves much like a multi-CPU system sharing memory with coherent caches (a “symmetric multiprocessor” or SMP system).

Virtual Processing Element :see VPE, next

VPE: one or more *TC*s sharing a bank of CP0 registers and privileged-architecture resources make up a VPE. The “Tera” project called this a “team”.

A single TC running in its own VPE - as seen by software unaware of the MIPS MT ASE - looks like an independent CPU compliant with the MIPS32/MIPS64 specifications. So you can run legacy software (including any OS) which is compatible with the MIPS architecture on a VPE even though the legacy software knows nothing about multi-threading.

Yield Qualifier:a signal presented to the core interface which is available for test by the **`yield`** instruction; see Section 2.8.1, "Yield, Yield Qualifiers and threads waiting for hardware events".

Appendix C

CP0 register summary and reference

This appendix lists all the CP0 registers of the 1004K core. You can find registers by name through Table B.1, by number through Table B.3 and there’s our best shot at functional groupings below in Table B.4. The registers-by-number Table B.3 tells you where to find a detailed description - if you’re reading on-line it’s a hot-link.

Power-up state of CP0 registers

The traditions of the MIPS architecture regard it as software’s job to initialize CP0 registers. As a rule, only fields where a wrong setting would prevent the CPU from booting are forced to an appropriate state by reset; other fields - including other fields in the same register - are random. This manual documents where a field has a forced-from-reset value; but your rule should be that all CP0 registers should be initialized unless you are quite sure that a random value will be harmless.

A note on unused fields in CP0 registers

Unused fields in registers are marked either with a digit 0 or an “X”. A field marked zero should always be written with zero, and subject to that is guaranteed to read zero on cores in the 1004K family. A field marked “X” may return any value, and nothing you write there will have any effect - but unless stated otherwise, it’s usually best to write it either as zero or with a value you previously read from it.

C.1 CP0 registers by name

Table C.1 Register Index by Name

Name	Number	Name	Number	Name	Number	Name	Number
BadVAddr	8.0	*EPC*	14.0	*PerfCtl0-3*	25.0	*TCStatus*	2.1
CacheErr	27.0	*ErrCtl*	26.0		25.2	*TraceControl*	23.1
Cause	13.0	*ErrorEPC*	30.0		25.4	*TraceControl2*	23.2
CDMMBase	15.2	*HWREna*	7.0		25.6	*TraceDBPC*	23.5
CMGCRBase	15.3				15.0	*TraceIBPC*	23.4
Compare	11.0	*Index*	0.0	*PRId*			
Config	16.0	*IntCtl*	12.1	*Random*	1.0	*UserLocal*	4.2
Config1-2	16.1-2	*IDataHi*	29.1	*SRSConf0-4*	6.1-5	*UserTraceData*	23.3
Config3	16.3	*IDataLo*	28.1	*SRSCtl*	12.2	*VPEConf0*	1.2
Config7	16.7	*ITagLo*	28.0	*SRSMap*	12.3	*VPEConf1*	1.3
Context	4.0	*L23DataHi*	29.5	*Status*	12.0	*VPEControl*	1.1
Count	9.0	*L23DataLo*	28.5	*TCBind*	2.2	*VPEOpt*	1.7
Debug	23.0	*L23TagLo*	28.4	*TCContext*	2.5	*WatchHi0-3*	19.0-3
DEPC	24.0	*LLAddr*	17.0	*TCHalt*	2.4	*WatchLo0-3*	18.0-3
DESAVE	31.0	*MVPConf0-1*	0.2-3	*TCRestart*	2.3	*Wired*	6.0
DDataLo	28.3	*MVPControl*	0.1	*TCSchedule*	2.6	*YQMask*	1.4
DTagLo	28.2	*PageMask*	5.0	*VPESchedule*	1.5		
EBase	15.1	*PerfCnt0-3*	25.1	*TCScheFBack*	2.7		
EntryHi	10.0		25,3				
EntryLo0-1	2.0		25.5				
	3.0		25.7				

C.2 CP0 registers by number

In a MIPS MT CPU almost all CP0 registers are replicated per-VPE. But registers whose name starts with “TC” are replicated per-TC. Then the two “MVP...” registers and the performance counters are global (there’s only one of each of these registers per CPU). Note that there are a few fields in other registers which don’t fit in with this: see Section 2.9.1 “What CP0 registers are per-TC, per-VPE and per-CPU?” for the gory details.

Table C.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
0.0	*Index*	Index into the TLB array	5.8.4, p.90
0.1	*MVPControl*	CPU-wide multithreading control	Figure 2.5 , p. 38
0.2-3	*MVPConf0-1*	CPU’s multithreading resources	Figure 2.4 , p. 37

Table C.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
1.0	*Random*	Randomly generated index into the TLB array	5.8.4, p.90
1.1	*VPEControl*	VPE control and status	Figure 2.1 , p. 34
1.2-3	*VPEConf0-1*	Initializable per VPE resource lists	Figure 2.6 , p. 39
1.4	*YQMask*	Defines valid inputs for **`yield`** instruction	2.9.9, p.39
1.5	*VPESchedule*	Per-VPE thread policy hints	2.9.12, p.40
1.6	*VPEScheFBack*	Per-VPE information from policy manager	
1.7	*VPEOpt*	Per-VPE cache-way inhibition	2.9.10, p.40
2.0	*EntryLo0*	Output (physical) side of TLB entry for even-numbered virtual pages	Figure 5.13 , p. 91
2.1	*TCStatus*	Status and control for each TC	2.9.4, p.35
2.2	*TCBind*	VPE affiliation and own TC number of this TC	Figure 2.3 , p. 37
2.3	*TCRestart*	Where this TC will next fetch code from	2.9.3 , p. 35
2.4	*TCHalt*	Set 1 to freeze the TC for inspection/modification	2.9.3 , p. 35
2.5	*TCContext*	Read/write scratch register for OS to maintain thread ID	2.9.3 , p. 35
2.6	*TCSchedule*	Per-TC thread scheduling hints	2.9.12 , p. 40
2.7	*TCScheFBack*	Per-TC information from policy manager	
3.0	*EntryLo1*	Output (physical) side of TLB entry for odd-numbered virtual pages	Figure 5.13 , p. 91
4.0	*Context*	Mixture of pre-programmed and *BadVAddr* bits which can act as an OS page table pointer.	Figure 5.14 , p. 93
4.2	*UserLocal*	Kernel-writable but user-readable software-defined thread ID	C.4.2, p.192
5.0	*PageMask*	Control for variable page size in TLB entries	Figure 5.12 , p. 91
6.0	*Wired*	Controls the number of fixed (“wired”) TLB entries	5.8.4 , p. 90
6.1-5	*SRSConf0-4*	Write these to use TCs as shadow registers	Figure 7.5 , p. 113
7.0	*HWREna*	Select which hardware registers are readable using the **`rdhwr`** instruction in user mode.	Figure 7.6 , p. 115
8.0	*BadVAddr*	Reports the address for the most recent TLB-related exception	5.8.7, p.93
9.0	*Count*	Free-running counter at pipeline or sub-multiple speed	C.4.4, p.193
10.0	*EntryHi*	High-order portion of the TLB entry	Figure 5.12 , p. 91
11.0	*Compare*	Timer interrupt control	C.4.4, p.193
12.0	*Status*	Processor status and control	Figure C.1 , p. 190
12.1	*IntCtl*	Setup for interrupt vector and interrupt priority features.	Figure 7.1 , p. 107
12.2	*SRSCtl*	Shadow register set selectors	Figure 7.3 , p. 111
12.3	*SRSMap*	In VI (vectored interrupt) mode, determines which shadow set is used for each interrupt source.	Figure 7.4 , p. 113
13.0	*Cause*	Cause of last general exception	Figure C.2 , p. 192
14.0	*EPC*	Restart address from exception (no subfields, not described further in this manual)	[MIPS32]
15.0	*PRId*	Processor identification and revision	Figure 4.6 , p. 61
15.1	*EBase*	Exception entry point base address and CPU/VPE ID	Figure 7.2 , p. 110

Table C.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
15.2	*CDMMBase*	Base address for common device memory map region	Figure 5-9 , p. 88
15.3	*CMGCRBase*	Base address for global configuration registers	
16.0	*Config*	Configuration register	Figure 4.1 , p. 56
16.1-2	*Config1-2*	Configuration for MMU, caches etc	Figure 4.2 , p. 58
16.3	*Config3*	Interrupt and ASE capabilities	Figure 4.4 , p. 59
16.7	*Config7*	1004K family-specific configuration	Figure C-3 , p. 194
17.0	*LLAddr*	Address associated with last `ll` instruction of the “load-linked/store-conditional” instruction pair. Not used in normal OS code.	3.5 , p. 52
18.0-3	*WatchLo0-3*	Watchpoint address: *WatchLo0-1* are I-side, and *WatchLo2-3* are D-side	Figure 10.22 , p. 171
19.0-3	*WatchHi0-3*	Watchpoint control: again, *WatchHi0-1* are I-side, and *WatchHi2-3* are D-side	
23.0	*Debug*	EJTAG Debug register	Figure 10.1 , p. 148
23.1	*TraceControl*	Control fields for the PDTrace unit.	Figure 10.18 , p. 165
23.2	*TraceControl2*		
23.3	*UserTraceData*	Software-generated PDTrace information register	10.2.4 , p. 168
23.4	*TraceBPC*	Additional controls for PDTrace start/stop	Figure 10.21 , p. 168
24.0	*DEPC*	Restart address from last EJTAG debug exception	10.1.7 , p. 147
25.0 25.2 25.4 25.6	*PerfCtl0-3*	Performance counter control	Figure 10.24 , p. 172
25.1 25.3 25.5 25.7	*PerfCnt0-3*	Performance counters	
26.0	*ErrCtl*	Software parity control and test modes for cache RAM arrays	Figure 5.7 , p. 84
27.0	*CacheErr*	Cache parity exception control and status	5.4.17, p.82
28.0	*ITagLo*	Cache tag read/write interface for I-, D- and L2 (secondary) cache respectively	5.2, p.81 C.4.7, p.196
28.2	*DTagLo*		
28.4	*L23TagLo*		
28.1	*IDataLo*	Low-order data read/write interface for I-, D- and L2 cache respectively...	
28.3	*DDataLo*		
29.1	*IDataHi*	... and high-order data for the I-cache, which is only accessible in 64-bit units.	
28.5	*L23DataLo*	Read/write data for L2 cache	
29.5	*L23DataHi*	Read/write check bits (ECC) for L2 cache	
30.0	*ErrorEPC*	Restart location from a reset or a cache error exception	5.4.17, p.82
31.0	*DESAVE*	Scratch read/write register for EJTAG debug exception handler	10.1.7, p.147

C.3 CP0 registers by function

Table C.3 CP0 registers grouped by function

Basic modes	*Status*	12.0	**Cache Management**	*DDataLo*	28.3	**Control user rdhwr access**	*HWREna*	7.0
Exception control	*Cause*	13.0		*DTagLo*	28.2	**Parity/ECC control**	*CacheErr*	27.0
	EPC	14.0		*ErrCtl*	26.0	**Multithreading (global)**	*MVPConf0-1*	0.2-3
OS/userland thread ID	*UserLocal*	4.2		*ErrorEPC*	30.0		*MVPControl*	0.1
				IDataHi	29.1	**Multithreading (per-TC)**	*TCBind*	2.2
Timer	*Compare*	11.0		*IDataLo*	28.1		*TCContext*	2.5
	Count	9.0		*ITagLo*	28.0		*TCHalt*	2.4
CPU Configuration	*CDMMBASE*	15.2		*L23DataHi*	29.5		*TCRestart*	2.3
	CMGCRBase	15.3						
	Config	16.0		*L23DataLo*	28.5		*TCScheFBack*	2.7
	Config1-2	16.1-2		*L23TagLo*	28.4		*TCSchedule*	2.6
	Config3	16.3	**EJTAG debug**	*DEPC*	24.0		*TCStatus*	2.1
	Config7	16.7		*DESAVE*	31.0		*VPESchedule*	1.5
	EBase	15.1		*Debug*	23.0	**Multithreading (per-VPE)**	*SRSConf0-4*	6.1-5
	IntCtl	12.1	**PDtrace block**	*TraceControl*	23.1		*TCSchedule*	2.6
	PRId	15.0		*TraceControl2*	23.2		*VPEConf0*	1.2
	SRSCtl	12.2		*TraceDBPC*	23.5		*VPEConf1*	1.3
	SRSMap	12.3		*TraceIBPC*	23.4		*VPEControl*	1.1
TLB Management	*BadVAddr*	8.0		*UserTraceData*	23.3		*VPEOpt*	1.7
	Context	4.0	**debug/analysis**	*WatchHi0-3*	19.0-3		*VPESchedule*	1.5
	EntryHi	10.0		*WatchLo0-3*	18.0-3		*YQMask*	1.4
	EntryLo0-1	2.0	**Profiling**	*PerfCnt0-3*	25.1	**Multithreading configuration**	*MVPConf0-1*	0.2-3
		3.0			25.3		*SRSConf0-4*	6.1-5
	Index	0.0			25.5		*TCBind*	2.2
	PageMask	5.0			25.7		*VPEConf0*	1.2
	Random	1.0		*PerfCtl0-3*	25.0		*VPEConf1*	1.3
	Wired	6.0			25.2		*VPEOpt*	1.7
					25.4			
					25.6			

C.4 Miscellaneous CP0 register descriptions

Many CP0 registers in the 1004K core are already described earlier in this manual, in a relevant section. But those which got missed are described below, to make sure that every CP0 register field is at least mentioned in this manual.

C.4.1 Status register

The *Status* register is the most basic (and most diverse, for historical reasons) control register in the MIPS architecture, and its fields are squashed into Figure C.1. All fields are writable unless noted otherwise.

Figure C.1 All Status register fields

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 10	9 8	7	6	5	4 3	2	1	0
CU3-0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	CEE	0	IM7-0		KX	SX	UX	KSU	ERL	EXL	IE
In EIC (external int controller) mode													IPL	IM1-0							

The 1004K family *Status* has no non-standard fields - they're all as defined by [MIPS32]. Here and elsewhere these field descriptions are fairly terse, and you should read behind this if you're new to the MIPS architecture. Few of the fields in *Status* are guaranteed to be initialized by hardware on a CPU reset; bootstrap code should write a reasonable value to it early on (the same is true of many other CP0 registers, and the rule is "unless you know it's safe to leave it random, initialize it").

A few fields are somewhat core-specific, and they are described at more length.

CU3-0: enables for different co-processor instruction sets (replicated per-TC). Writable when such a coprocessor exists. Since no 1004K family CPU has a co-processor3, *Status[CU3]* is hard-wired zero.

Setting *Status[CU0]* to 1 has the peculiar effect of allowing privileged instructions to work in user mode; not something a secure OS is likely to allow often.

RP: Reduced power - standard field.

It's not connected inside the 1004K core, but the state of the RP bit is available on the external core interface as the *SI_RP* signal. The 1004K core uses clocks generated outside the core, and this could be used in your design to slow the input clock(s).

FR: if there is a floating point unit, set 0 for MIPS I compatibility mode (which means you have only 16 real FP registers, with 16 odd FP register numbers reserved for access to the high bits of double-precision values).

RE: reverse endianness in user mode. Hard-wired to zero in the 1004K core, which doesn't provide this feature.

MX: write 1 to enable instructions in *either* the MIPS DSP extension to the MIPS architecture, *or* the MDMX™ extension. The two may not be used together, so MDMX will never be available for the 1004K core. But for maximum portability you can find out which by looking at *Config3[DSPP]* (1 if MIPS DSP is implemented) and *Config1[MD]* (1 if MIPS MDMX is implemented).

PX: see description of *UX* below (but always zero on the 32-bit 1004K CPU).

BEV: "boot exception vectors" - when 1, relocates all exception vectors to near the reset-time start address. See Section 7.3.1 "Summary of exception entry points". This bit is automatically set when the CPU is reset.

TS: (read-only) records whether there has been any "machine check" exception (caused by duplicate valid TLB entries, generally a rather serious error) since the CPU was reset. Will always read 0 on 1004K core.

SR: MIPS32 architecture "soft reset" bit: the 1004K core’s interface only supports a full external reset, so this always reads zero.

NMI: (read-only) - non-maskable interrupt shares the "reset" handler code, this field reads 1 when it was a NMI event which caused it.

CEE: CorExtend Enable: read/write bit. Set zero to disable "CorExtend" user-defined instructions.

Not all CorExtend blocks implement this bit (those that don’t are unconditionally enabled). But CorExtend blocks should use this facility if they store internal state and rely on the OS to save/restore the state associated with some particular task. In such blocks, running a CorExtend instruction with *Status[CEE]* set to zero will cause the CPU to take a “CorExtend Unusable” exception - *Cause[ExcCode]* value 17. A suitably aware kernel will catch the exception and use it to note that the task is one which uses CorExtend resources (and therefore will need CorExtend state saved and restored appropriately).

Do not attempt to set this bit if CorExtend is not present.

IM7-0: bitwise interrupt enable for the eight interrupt conditions also visible in *Cause[IP7-0]*; *except* in the "EIC" interrupt mode, see Section 7.2.3 “External Interrupt Controller (EIC) mode”. In that case (as shown) the upper six bits become the “interrupt priority level” (“IPL”) value in the range 0-63.

KX,SX,UX: the MIPS architecture’s memory mapping system changes slightly to support 64-bit addressing, and these bits make that change for kernel-, supervisor- and user-privilege code respectively. But the 1004K core is a 32-bit CPU, so these are always zero.

KSU: execution privilege level - basically user or kernel:

0	kernel
1	supervisor
2	user

Now that the intermediate “supervisor” privilege level is rarely used, this field is often shown as two separate bits, with the bit 4 being called *UM* (“1 for user mode”).

ERL: "cache parity error exception mode" - which is really a stronger version of the exception mode *Status[EXL]* bit whose description follows...

EXL: exception mode bit, set automatically when you first enter an exception handler or upon reset (reset is treated like an exception). MIPS hardware barely supports nested exceptions, so this disables interrupts and software should avoid causing an exception in the early part of the handler[41].

IE: global interrupt enable, 0 to disable all interrupts.

[41] There are some very special cases where nested exceptions are permitted, and the architecture specifies some rather special behaviors to support those. But they’re beyond the scope of this manual; see [SEEMIPSRUN]:: or the [MIPS32] bible.

C.4.2 The *UserLocal* register

Not interpreted by hardware, this register is suitable for a kernel-maintained thread ID whose value can be read by user-level code with **`rdhwr $29`**, so long as *HWREna[UL]* is set.

In multithreading CPUs, *UserLocal* is replicated per-TC, and the **`fork`** instruction copies the parent's *UserLocal* value to the child's.

UserLocal was a late addition to the architecture. Kernels should check whether this register is implemented by inspecting *Config3[ULRI]*, as described in Section 4.1.3 "The Config3 register". Use of **`rdhwr $29`** will cause an exception in CPUs not implementing this register, providing an opportunity for an OS kernel to simulate it.

C.4.3 Exception handling: Cause register

The Cause register is the first thing to consult after you get an exception, to figure out why the exception happened (and therefore, what to do about it):

Figure C.2 Fields in the Cause register

31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP7-2	IP1-0	0	ExcCode	0
								In EIC (external int controller) mode	RIPL				

Cause tells you about the exception which just happened. Most fields are read-only:

BD: 1 if the exception happened on an instruction in a branch delay slot; in this case *EPC* is set to restart execution at the branch, which is usually the correct thing to do. You need only consult *Cause[BD]* when you need to look at the instruction which caused the exception (perhaps to emulate it).

TI: last interrupt was from the on-core timer (see section below for *Count*/*Compare*.)

CE: if that was a "co-processor unusable" exception, this is the co-processor which you tried to use.

DC: (writable) set 1 to disable the *Count* register.

PCI: last interrupt was an overflow from the performance counters, see Section 10.4 "Performance counters".

IV: (writable) set 1 to use a special exception entry point for interrupts, see Section 7.3.1 "Summary of exception entry points". It's quite likely that if you're doing this, you're also using multiple entry points for different interrupt levels; see Section 7.2 "MIPS32® Architecture Release 2 - enhanced interrupt system(s)".

WP: (writable to zero) - remembers that a watchpoint triggered when the CPU couldn't take the exception because it was already in exception mode (or error-exception mode, or debug mode). Since this bit automagically causes the exception to happen again, it must be cleared by the watchpoint exception handler.

IP7-0, RIPL: the current state of the interrupt request inputs. When one of them is active and enabled by the corresponding *Status[IM7-0]* bit, an interrupt may occur.

IP1-0 are writable, and in fact always just reflect the value written here. They act as software interrupt bits.

When using "EIC" interrupt mode the interpretation of this field changes, hence the alternate name of *RIPL* ("requested interrupt priority level"). In EIC mode this represents a value between 0 and 63, and reflects the code presented on the incoming interrupt lines when the exception happened. For more information see Section 7.2.3 "External Interrupt Controller (EIC) mode".

ExcCode: what caused that last exception. Lots of values, listed in Table C.4.

Table C.4 Exception Code values in *Cause[ExcCode]*

Val	Code	What just happened?
0	`Int`	Interrupt
1	`Mod`	Store, but page marked as read-only in the TLB
2	`TLBL`	Load or fetch, but page marked as invalid in the TLB
3	`TLBS`	Store, but page marked as invalid in the TLB
4	`AdEL`	Address error on load/fetch or store respectively. Address is either wrongly aligned, or a privilege violation.
5	`AdES`	
6	`IBE`	Bus error signaled on instruction fetch
7	`DBE`	Bus error signaled on load/store (imprecise)
8	`Sys`	System call, ie **`syscall`** instruction executed.
9	`Bp`	Breakpoint, ie **`break`** instruction executed.
10	`RI`	Instruction code not recognized (or not legal)
11	`CpU`	Co-processor instruction encoding for co-processor which is not enabled in *Status[CU3-0]*.
12	`Ov`	Overflow from trapping form of integer arithmetic instructions.
13	`Tr`	Condition met on one of the conditional trap instructions **`teq`** etc.
14	-	Reserved
15	`FPE`	Floating point unit exception - more details in *FCSR*.
16	-	Available for implementation dependent use
17	CeU	CorExtend instruction attempted when not enable by *Status[CEE]*
18	`C2E`	Reserved for precise Coprocessor 2 exceptions
19-21	-	Reserved
22	`MDMX`	Tried to run an MDMX instruction but *Status[MX]* wasn't set (most likely, the CPU doesn't do MDMX)
23	`WATCH`	Instruction or data reference matched a watchpoint
24	`MCheck`	"Machine check" - never happens in the 1004K core.
25	`Thread`	Thread-related exception, as described in [MIPSMT]; there's a sub-cause field in *VPEControl[EXCPT]*, as shown in Figure 2.1.
26	`DSP`	Tried to run an instruction from the MIPS DSP ASE, but it's not enabled (that is, *Status[MX]* is zero).
27-29	-	Reserved
30	`CacheErr`	Parity/ECC error somewhere in the core, on either instruction fetch, load or cache refill. In fact you never see this value in *Cause[ExcCode]*; but some of the codes in this table including this one can be visible in the "debug mode" of the EJTAG debug unit - see Section 10.1 "EJTAG on-chip debug unit", and in particular the notes on the *Debug* register in Figure 10.1.
31	-	Reserved

C.4.4 Count and Compare

These two 32-bit registers form a useful and flexible timer. *Count* just counts. For the 1004K core, that's usually at the full pipeline clock rate. But portable software can discover how fast *Count* counts by reading the "hardware register" called "CCRes", see Section 6.1 "User-mode accessible "Hardware registers"".

You can write *Count* to set a value in it, but it’s generally more valuable for an OS to leave it as a free-running counter.

When the value of *Count* coincides with the value in *Compare*, an interrupt is raised. The interrupt is cleared every time *Compare* is written. This is handy:

- For a periodic interrupt, simply advance *Compare* by a fixed amount each time (and check for the possibility that *Count* has overrun it).
- To set a timer for some point in the future, just set *Compare* to an increment more than the current value of *Count*.

The timer interrupt is implemented as an output signal at the core interface; but it’s conventional (well, pretty compulsory if you want OS’ to work) to return it to the CPU core on an interrupt line — to the same VPE, since the two VPEs have separate interrupt inputs. Your system integrator should wire information pins to show where the timer interrupt is connected: see the notes on *IntCtl[IPTI]* below Figure 7.1. However, if you have an “EIC” interrupt controller (see Section 7.2.3 “External Interrupt Controller (EIC) mode”) you’ll need to send the timer interrupt all the way out to the interrupt controller and back.

C.4.5 The Config7 register

The *Config7* register holds CPU-specific one-time setup and basic information fields.

Figure C-3 Fields in the Config7 Register

31	30 20	19	18	17	16	15 11	10	9	8	7	6	5	4	3	2	1	0
WII	0	PCT	HCI	FPR	AR	0	IAR	IVA	ES	BTLM	0	NBLSU	ULB	BP	RPS	BHT	SL
			0					0	0	0	—	0	0	0	0	0	0

Config7: read-only fields

Config7[WII]: Read-only bit which tells you how **`wait`** behaves. When this bit is set, an interrupt which would occur just so long as *Status[IE]* is set 1 and *TCStatus[IXMT]* is cleared to 0 will always be enough to terminate a **`wait`** instruction.

1004K family CPUs where *WII* reads 0 will remain in the wait condition forever if entered with interrupts disabled.

The MIPS32 Architecture Specification permits either behavior.

But with the *WII*-set feature it’s safe to **`wait`** with interrupts disabled using *Status[IE]* or *TCStatus[IXMT]*. This allows OS code to avoid a tricky race condition.

Config7[PCT]: read-only field which reads 1 if the core has two performance counters, replicated per-TC (older cores had four performance counters, not replicated): see Section 10.4 “Performance counters”.

Config7[HCI]: read-only field which is always zero on 34K family cores. It reads 1 for some software-simulated CPUs, to indicate that the software-modelled cache does not require initialization. Most software should ignore this bit.

Config7[FPR]: read-only field. Reads 1 if an FPU is fitted but (as is common) it runs at half the main core clock rate.

Config7[AR]: read-only field, indicating that the D-cache is configured to avoid *cache aliases*.

Config7[IAR]: a read-only field which tells you that you have an I-cache whose cacheops can be made alias-proof, as described in Section 5.4.10 “Cache aliases”.

Config7: writable fields

Config7[IVA]: is hard-wired zero when the cache is inherently alias-free, as when the cache size is 16KB or less. Otherwise this field can be used to enforce legacy behaviour on a CPU which has “alias-proof” I-cache cacheops — see *Config7[IAR]* field above.

Config7[ES]: when it is set to "1", the **sync** instruction will be signalled on the core’s OCP interface as an "ordering barrier" transaction, using a **sync**-specific encoding. It defaults to zero at system reset

Config7[ES] bit cannot be set (will always read zero and will have no effect) unless the OCP input signal *SI_SyncTxEn* is asserted — it’s interpreted as agreement from the connected OCP device/interconnect that it can handle the barrier transaction.

In this multithreading CPU this option may be set *only* for the whole CPU: setting it for one VPE sets it for the other.

Config7[BTLM]: Set this bit to enable a schedule feature, where any TC which has an unresolved load miss pending will automatically drop in scheduling priority (below any non-load-blocked runnable TC). Thread scheduling is described in Section 3.2, "Thread scheduling in the 1004K™ core".

The remaining fields default to zero and are uncommonly set. It is therefore always safe *not* to write *Config7*. Some of these bits are for diagnostics and experimentation only:

Config7[NBLSU]: set 1 to arrange that load/store pipeline stalls will stop the main pipeline too, keeping them synchronized. For debug and investigation only.

Config7[ULB]: set 1 to make all uncached loads blocking (a program usually only blocks when it uses the data which is loaded). You want to do this only when nothing else will work...

Config7[BP]: when set, no branch prediction is done, and all branches and jumps cause instruction fetch to be suspended until they are resolved.

Config7[RPS]: when set, the return address branch predictor is disabled, so **jr $31** is treated just like any other jump register. Instruction fetch stalls after the branch delay slot, until the jump instruction reaches the "EX" stage in the pipeline and can provide the right address (typically adds 5 clocks compared to a successfully predicted return address).

Config7[BHT]: when set, the branch history table is disabled and all branches are predicted taken.

Config7[SL]: when set, disables non-blocking loads. Normally the 34K core will keep running after a load instruction even if it misses in the D-cache, until the data is used. With this disable bit set, the CPU will stall on any load D-cache miss.

C.4.6 The *CMGCRBase* register

This read-only register reports the base physical address for the Global Control Registers located in the Coherence Manager. The presence of this register is indicated by the *Config3[CMGCR]* bit.

Figure C-4 Fields in the CMGCRBase Register

31 … 11	10 … 0
CMGCR_BASE_ADDR	0

CMGCRBase[CMGCR_BASE_ADDR]: Base Address for the Global Control Registers. The address is shifted to allow a 36b PA in the register (bits 31:11 correspond to PA[35:15]). SW will need to unshift it and generate a kseg1 or mapped virtual address that will reach the phyiscal address. But even this is an improvement over just needing to know where it is.

C.4.7 Cache registers in special diagnostic modes

Most of the way that cache tag registers work is common (to a large extent) over most recent MIPS Technologies cores. Those common features are described in Section 5.4.12 "Cache initialization and tag/data registers". More obscure features are here.

DTagLo, ITagLo registers when accessing Way Select RAM

This is the view you get when *ErrCtl[WST]* is set.

Figure C-5 Fields in the TagLo Register (*ErrCtl[WST]* set)

31 … 24	23 … 20	19 … 16	15 … 10	9 … 8	7 … 5	4 … 1	0
U	WSDP	WSD	LRU	0	U	0	U

TagLo-WST[WSD,WSDP]: cache line dirty bits are held in the "way select" RAM, to make them easier to update. Here you can see all of them, and each has a parity bit.

TagLo-WST[LRU]: when you read or write the tag in way select test mode (that is, with *ErrCtl[WST]* set) this field reads or writes the LRU ("least recently used") state bits, held in the way select RAM.

Appendix D

MIPS® Architecture quick-reference sheet(s)

D.1 General purpose register numbers and names

By ancient convention the general-purpose registers in the MIPS architecture have conventional names which remind you of their standard usage in popular MIPS ABIs. Table D.1 shows those names related to both the “o32” ABI (almost universally used for 32-bit MIPS applications), but also the minor variations in the “n32” and “n64” ABIs defined by Silicon Graphics.

If you’re not sure what an ABI is, just read the “o32” column!

Table D.1 Conventional names of registers with usage mnemonics

Register Nos	*name*	*use*		
`$0`	`zero`	always zero		
`$1`	`AT`	assembler temporary		
`$2-$3`	`v0-v1`	return value from function		
`$4-$7`	`a0-a3`	arguments		
	o32		*n32/n64*	
	name	*use*	*name*	*use*
`$8-$11`	`t0-t3`	temporaries	`a4-a7`	more arguments
`$12-$15`	`t4-t7`		`t0-t3`	temporaries
`$24-$25`	`t8-t9`		`t8-t9`	
`$16-$23`	`s0-s7`	saved registers		
`$26-$27`	`k0-k1`	reserved for interrupt/trap handler		
`$28`	`gp`	global pointer		
`$29`	`sp`	stack pointer		
`$30`	`s8/fp`	frame pointer if needed (additional saved register if not)		
`$31`	`ra`	Return address for subroutine		

D.2 User-level changes with Release 2 of the MIPS32® Architecture

With the Release 2 update the MIPS32 instruction set gains some useful extra features, shown below. User-level programs also get limited access to “hardware registers”, useful for user-privilege software but which wants to adapt (portably) to get the best out of the CPU.

D.2.1 Release 2 of the MIPS32® Architecture - new instructions for user-mode

The following instructions are new with the MIPS32 release 2 update:

Table D.2 Release 2 of the MIPS32® Architecture - new instructions

Instruction(s)	Description

`ehb`
`jalr.hb rd, rs`
`jr.hb rs`
	Hazard barriers; wait until side-effects from earlier instructions are all complete (that is, can be guaranteed to apply in full to all instructions issued after the barrier).
	These defend you respectively against:
	`ehb` - execution hazards (side-effects of old instructions which affect how an instruction executes, but excluding those which affect the instruction fetch process).
	`jalr.hb`/`jr.hb` - hazards of all kinds.
	Note that `eret` is also a barrier to all kinds of hazard.

`ext rt, rs, pos, size`
`ins rt, rs, pos, size`
	Bitfield extract and insert operations.

`mfhc1 rt, fs`
`mthc1 rt, fs`
	Coprocessor/general register move instructions targeting the high-order bits of a 64-bit floating point unit (CP1) register when the integer CPU is 32-bit.

`mfhc2 rt, rd`
`mthc2 rt, rd`
	Coprocessor2 might be 64 bits, too (but this is typically a customer special unit).

`rdhwr rt,rd`	"read hardware register" - user-mode access read-only access to low-level CPU information - see "Hardware Registers" below.

`rotr rd, rt, sa`
`rotrv rd, rt, rs`
	Bitwise rotate instructions (like shifts, one has the rotate amount as an immediate field `sa`, the other in an additional register argument `rs`).

`seb rd, rt`
`seh rd, rt`
	Register-to-register sign extend instructions.

`synci offset(base)`	Synchronize caches to make instruction write effective. Instructions written by the CPU for itself to execute must be written back from the D-cache and any stale data at that location invalidated from the I-cache, before it will work properly. `synci` is a user-privilege instruction which does all that is required for the enclosing cache-line sized memory block. Very useful to JIT interpreters.
`wsbh rd, rt`	swap the bytes in each halfword within a 32-bit word. It was introduced together with the rotate instructions `rot`/ `rotv` and the sign-extenders `seb`/ `seh`. Between them you can make big savings on common byte-twiddling operations; for example, you can swap the bytes in `$2` using `rot$2,$2,16; wsbh$2,$2`.

D.2.2 Release 2 of the MIPS32® Architecture - Hardware registers from user mode

The hardware registers provide useful information about the hardware, even to unprivileged (user-mode) software, and are readable with the `rdhwr` instruction. [MIPS32] defines four registers so far. The OS can control access to each register individually, through a bitmask in the CP0 register *HWREna* - (set bit 0 to enable register 0 etc). *HWREna* is cleared to all-zeroes on reset, so software has to explicitly enable user access. Privileged code can access any hardware register.

The five registers are:

- *CPUNum (0)*: Number of the CPU on which the program is currently running. This comes directly from the coprocessor 0 *EBase[CPUNum]* field.

- *SYNCI_Step (1)*: the effective size of an L1 cache line[42]; this is now important to user programs because they can now do things to the caches using the `synci` instruction to make instructions you've written visible for execution. Then *SYNCI_Step* tells you the "step size" - the address increment between successive `synci`'s required to cover all the instructions in a range.
  If *SYNCI_Step* returns zero, that means that you don't need to use `synci` at all.

[42] Strictly, it's the lesser of the I-cache and D-cache line size, but it's most unusual to make them different.

- *CC (2)*: user-mode read-only access to the CP0 *Count* register, for high-resolution counting. Which wouldn't be much good without...
- *CCRes (3)*: which tells you how fast *Count* counts. It's a divider from the pipeline clock (if the rdhwr instruction reads a value of "2", then *Count* increments every 2 cycles, at half the pipeline clock rate).
- *UserLocal (29)*: Scratch register of sorts. The kernel can store a thread specific value such as a thread ID or a pointer to thread specific storage to the underlying Cop0 register and user mode programs can read it via **`rdhwr`**

D.3 FPU changes in Release 2 of the MIPS32® Architecture

The main change is that a 32-bit CPU (like the 1004K CPU) can now be paired with a 64-bit floating point unit. The FPU itself is compatible with the description in [MIPS64V2].

The only new feature of the instruction set are the **`mfhc1`**/**`mthc1`** instructions described in Section D.2, "Release 2 of the MIPS32® Architecture - new instructions".

But it's worth stressing that the floating point unit implements 64-bit load and store instructions. The FPU of the 1004K CPU is described in Chapter 8, "Floating point unit" on page 117.

Appendix E

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Revision	Date	Description
1.00	20th June 2008	• Initial version
1.01	19th December 2008	• Added example idle loop code making use of *Config7[WII]* • Clarified duplicate TLB entry section
1.10	July 15, 2009	• Added example of mftr/mttr blocking on access to unhalted TC • Added coherent CCAs to cache code table • Added new performance counter events • Added FastDebugChannel description • Renumbered HW breakpoint registers in DRSEG table to match other docs (0..15 rather than 1..16) • New relocatable debug exception entry point • Mention PC sampling extensions • Newer PDtrace version with memory mapped access to on-chip trace buffer
1.20	January 21, 2011	• Removed errant statement that supervisor mode was not supported