Programming the MIPS32® 24K® Core Family

Document Number: MD00355
Revision 04.63
December 19, 2008

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

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 97).

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 24K® or 24KE™ core. So this document concentrates on where a MIPS 24K 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 details are widely spread; so you'll find a shortform presentation of the changes here in Section C.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 24K core.

All 24K cores are able to run programs encoded with the MIPS16e™ instruction set extension - which makes the binary significantly smaller, with some trade-off in performance. MIPS16e code is rarely seen - it's almost exclusively produced by compilers, and in a debugger view is pretty much a subset of the regular MIPS32 instruction set - so you'll find no further mention of it in this manual; please refer to [MIPS16e].

The document is arranged functionally: very approximately, the features are described in the order they'd come into play in a system as it bootstraps itself and prepares for business. But a lot of the CPU-specific data is presented in co-processor zero ("CP0") registers, so you'll find a cross-referenced list of 24K core CP0 registers in Appendix B, "CP0 register summary and reference" on page 99.

1. References (in square brackets) are listed in Chapter A, "References" on page 97.

1.1 Chapters of this manual

- Chapter 2, “Initialization and identity” on page 17: what happens from power-up? boot ROM material, but a good place to cover how you recognize hardware options, configure software-controlled ones and recognize your CPU..
- Chapter 3, “Memory map, caching, reads, writes and translation” on page 23: everything about memory accesses.
- Chapter 4, “Programming the 24K® core in user mode” on page 43: features relevant to user-level programming; multiply timing, hardware registers, prefetching.
- Chapter 5, “Kernel-mode (OS) programming and Release 2 of the MIPS32® Architecture” on page 51: 24K-core-specific information about privileged mode programming.
- Chapter 6, “Floating point unit” on page 61: the 24K core’s floating point unit, available on models called 24Kf™.
- Chapter 7, “24K® core features for debug and profiling” on page 70: the debug and PDTrace™ units, plus separate watchpoints and performance counters..
- Appendix A, “References” on page 97: more reading to broaden your knowledge.
- Appendix B, “CP0 register summary and reference” on page 99: all the registers with links back into the main text.
- Appendix C, “MIPS® Architecture quick-reference sheet(s)” on page 111: basic CPU-independent information, including a quick description of Release 2 of the MIPS32 Architecture.

1.2 Typographical conventions

CPU register names are in *oblique monospace*. Co-processor 0 (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.

References to other manuals are collected together in Appendix A, “References” on page 97 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].

1.3 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! And note that not all register diagrams are color-coded yet.

- 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.

1.4 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 CP0 registers are listed in Appendix B, “CP0 register summary and reference” on page 99. That appendix has the registers listed by name (Table B.1), by number (Table B.2), and by function (Table B.3.) 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.

1.5 24K® core features

All 24K family cores conform to Release 2 of the MIPS32 architecture. You may have the following options:

- *I- and D-Caches*: 4-way set associative; may be 8Kbytes, 16Kbytes, 32Kbytes or 64Kbytes in size. 32Kbytes is likely to be the most popular; 64Kbyte caches will involve some cost in frequency in most processes.

  Optionally (but usually) the 32K and 64K D-cache configurations can be made free of “cache aliases” - see Section 3.4.8, "Cache aliases", which explains some software-visible effects. The option is selected when the “cache wrapper” is defined for the 24K core in your design and shows up as the *Config7[AR]* bit.

  Note that a 4-way set associative cache of 16Kbyte or less (assuming a 4Kbyte minimum page size) can’t suffer from aliases.

- *Floating point unit (FPU)*: if fitted, is a 64-bit unit (with 64-bit load/store operations), which most often runs at half the speed of the integer unit.

- *Fast multiplier*: 1-per-clock repeat rate for 32×32 multiply and multiply/accumulate.

- *The “CorExtend™” instruction set extension*: is available on 24KPro CPUs. [CorExtend] defines a hardware interface which makes it relatively straightforward to add logic to implement new computational (register-to-register) instructions in your CPU, using predefined instruction encodings. It’s matched by a set of software tools

which allow users to create assembly language mnemonics and C macros for the new instructions. But there's very little about the CorExtend ASE in this manual.

- *Optional Co-processor*: if your application requires special functions, or hardware implemented very close to the CPU, 24K cores define an interface to a customer-implemented "co-processor 2". This provides a great deal of freedom to define multiple new registers and instructions (though it will be quite a lot of work). The instruction set defines basic CP2 instructions (to access its registers, for loads/stores, and branch instructions which test an associated "condition bit".

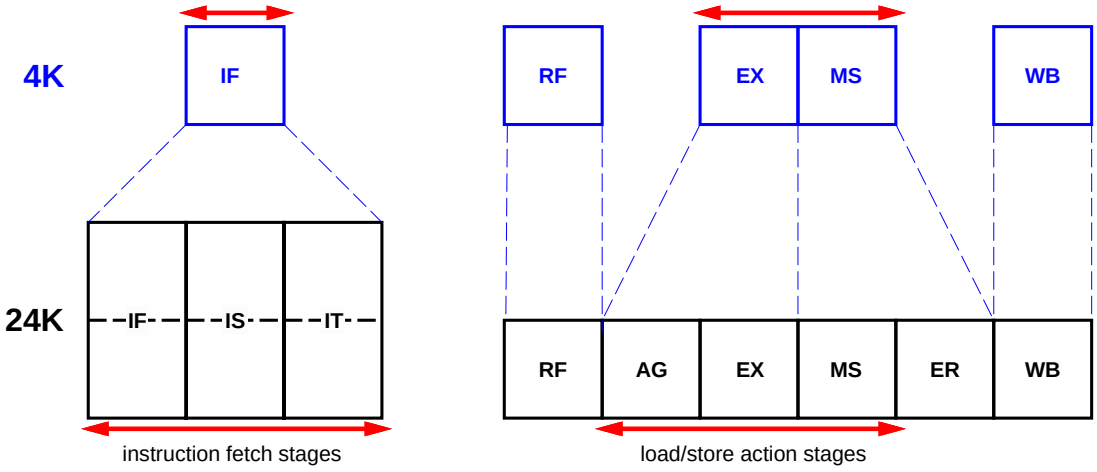
  CP2 implementations are far beyond the scope of this book. Talk to MIPS Technologies.

1.6 A brief guide to the 24K® core implementation

All 24K family cores are based on a nine-stage pipeline, where MIPS Technologies' first 4K™ family products had a five-stage pipeline: a simplified comparative diagram is at Figure 1.1. By reducing the amount of work to be done during each pipestage, the long pipeline allowed the design team to push up the operating frequency to a level unparalleled for a synthesizable design.

If you want to make a high-performance computer, there is no substitute for the highest frequency you can reach; but the longer pipeline makes it harder to keep issuing one instruction per clock (there are more instructions in flight which you might be dependent on). Long-pipeline CPUs can trip up on *branches* (they don't know where to fetch the next instructions until the branch instruction is substantially complete), and on *loads* (even on cache hits, the data cannot be available for some number of instructions); the 24K core is mainly different from its predecessors in the mechanisms used to mitigate those effects.

Figure 1.1 Pipeline differences between the 24K® and 4K™ core families



1.6.1 Notes on pipeline diagram (Figure 1.1):

Even in such a simplified diagram, there are a few points worth highlighting:

- *Longer cache access*: the extra pipeline stages are mostly used to give more time for access through the memory translation system (TLB) to the primary caches. Caches do not speed up quite so much as logic as the underlying chip geometry shrinks, and they've become the critical path at high frequency. Including address calculations, both I- and D-accesses are awarded three clocks.

- *Semi-detached instruction fetch unit*: the 24K core no longer has a single pipeline for most instructions; the instruction fetch unit (“IFU”) is semi-autonomous. It’s also 64 bits wide, and handles two instructions at a bite.

  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 the way you usually go. 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 section.

  The IFU has a queue to keep instructions in when it’s running ahead of the rest of the CPU. This kind of design is called a “decoupled” IFU.

- *Stretched load/store stages*: a dedicated address generation (“AG”) stage precedes the usual “EX” stage where arithmetic/logic operations happen, and the “MS” stage where the D-cache is accessed.

- *Slightly stretched arithmetic/logical operation time*: “EX” has to remain one stage so that dependent instructions can run next to each other without delay. But some logic can be pushed back into the new “AG” stage.

Now let’s focus on the 24K core’s mechanisms to ameliorate branch and load penalties.

1.6.2 Branches and branch delays

The MIPS architecture already defines that the instruction following a branch (the “branch delay slot” instruction) is always executed[2]. That means that the CPU has an extra instruction cycle time to figure out where a branch is going before suffering any delay. But with the 24K core’s long pipeline a branch instruction isn’t resolved until after the “EX” stage, five stages or so down the pipe; so a naive implementation would suffer at least a 4-clock penalty on every branch. Several different tricks are used:

- The decoupled IFU (the electronic dog) runs ahead of the rest of the CPU by fetching two instructions per clock. It can get as many as eight instructions ahead.

- Branch instructions are identified very early (in fact, they’re marked when instructions are fetched into the I–cache).

- The IFU’s *branch predictor* guesses whether conditional branches will be taken or not - it’s not magic, it uses a *Branch History Table* of what happened to branches in the past, indexed by the low bits of the location of the branch instruction. It makes no attempt to discover whether the “history” stored in a location is really that of the current branch, or another one which happened to share the same low bits; it’s harmless to be wrong sometimes. With a bit of cleverness which you could read about in [SUM], it guesses correctly most of the time.

  MIPS branches and jumps (at least those not dependent on register values) are easy to decode and the IFU decodes them locally. Then, armed with the taken/not-taken guess from the BHT, the IFU can predict the target address and continue to run ahead.

2. That’s not *quite* accurate: there are special forms of conditional branches called “branch likely” which are defined to execute the branch delay slot instruction only when the branch is taken. However, this was always meant to be done by allowing the branch delay instruction to run, then squishing it before it changes any machine state; and most implementations - including the 24K core - do it that way.
   Note that the “likely” part of the name has nothing to do with branch prediction; the 24K core’s branch prediction system treats the “likelies” just like any other branches.

In fact, the branch target calculation in the IFU is one clock too slow to guarantee a continuous stream of instructions: it's as if your dog takes a while to choose a path, and temporarily goes slower than you do. But so long as the dog was a few steps ahead of you to start with, you won't fall over it and it soon bounds ahead again.

- Jump-register instruction targets are unpredictable: the IFU has no knowledge of register data and can't in general anticipate it. But jump-register instructions are rare, except that...

  In the MIPS ISA you return from subroutines using a jump-register instruction, **`jr $31`** (register 31 is, by a strong convention, used to hold the return address). So on every call instruction, the IFU pushes the return address onto a small stack; and on every **`jr $31`** it pops the value of the stack and uses that as its guess for the branch target[3].

  On jump-register instructions using registers other than **`$31`** the IFU has to wait for the ALU to resolve the branch before it can continue.

- When the IFU guesses wrong, it doesn't know (the dog just rushes ahead until its owner reaches the fork).

  The mistake will be noticed once the branch instruction has proceeded down the pipeline to the "EX" stage, and is executed in its full context ("resolved"). The IFU tells the CPU what it did; if it turns out to be wrong the CPU must discard the instructions based on the guess (which fortunately will not have changed any vital machine state) and start fetching instructions from the correct target. The tug on the lead which goes out to the IFU is called a "redirect".

  Incorrect guesses (and unpredictable jumps such as a **`jr`** which is not to **`$31`**) are relatively expensive: four clocks are wasted.

1.6.3 Loads and load-to-use delays

Even short-pipeline MIPS CPUs can't deliver load data to the immediately following instruction without a delay, even on a cache hit. Simple MIPS pipelines typically deliver the data one clock later: a one clock "load-to-use delay". Compilers and programmers try to put some useful and non-dependent operation between the load and its first use.

The 24K core's long pipeline means that a full D-cache hit takes three clocks to return the data, not two. If (as in the 4K family) the memory access process started in the "EX" stage, that would lead to a two-clock load-to-use delay. But it's been found through painful experience that programmers and compilers find it much harder to find two non-dependent operations...

So the 24K core starts the memory access by doing initial address calculation in a new "AG" stage, before "EX". That keeps the load-to-use delay down to a sensible level. You'll hear this decision to defer the execute stage referred to as a "skewed ALU".

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 pointer chains is guaranteed to take at least three cycles per pointer.

3. The return-stack guess will be wrong for subroutines containing nested calls deeper than the size of the return stack; but subroutines high up the call tree are much more rarely executed, so this isn't so bad.

1.6.4 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 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 program has to be suspended while the updates are done.

Queues which can fill up include:

- *Cache refills in flight (four or eight)*: that's the size of the "FSB" queue - this and other queues are described in more detail under Section 3.3, "Reads, writes and synchronization". The CPU may run in parallel with a cache refill process because of its non-blocking loads, but usually only for a handful of instruction times. So you're unlikely to reach this limit unless you are using prefetch or otherwise deliberately optimizing loops. If a series of prefetches use enough available resources, the fourth outstanding load-miss will stall the pipeline. It's likely to be good practice for code making conscious use of prefetches to ration itself to two or three outstanding operations.

- *Non-blocking loads to registers (four or nine)*: there are just four entries in the "LDQ", each of which remembers one outstanding load, and which register the data is destined to return to. Compiled code is unlikely to reach this limit. If you write carefully optimized code where you try to fill load-use delays (perhaps for data you think will not hit in the D-cache) you may hit this problem.

- *Lines evicted from the cache awaiting writeback (4+)*: writes are collected in the "WBB" queue. The 24K core's ability to write data will in almost all circumstances exceed the bandwidth available to memory; so a long enough burst of writes will eventually slow to memory speed. There is probably nothing you can do about this.

Chapter 2

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.

2.1 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 24K core also defines *Config7* for some implementation-specific settings (which most programmers will never use).

Broadly speaking the registers have these roles:

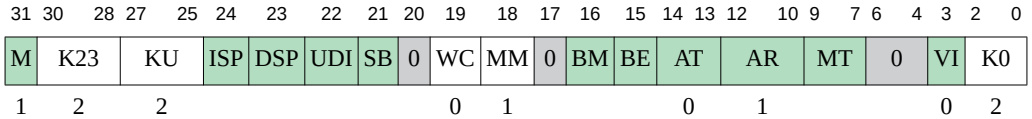
Table 2.1 Roles of *Config* registers

Config	A mix of historical and CPU-dependent information, described in Figure 2.1 below. Some fields are writable.
Config1 *Config2*	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 24K family cores). Shown in Figure 2.2 and Figure 2.3 below.
Config3	Read-only, strictly to Release 2 of the [MIPS32] architecture. More CPU capability information.
Config7	24K-core-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 B.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 5.2 and the text round it.

2.1.1 The Config register

Figure 2.1 Fields in the Config Register



In Figure 2.1:

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 3.3 in Section 3.4.2 “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 24K core 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 24K core-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 3.6, "Scratchpad memory/ SPRAM".

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

UDI: reads 1 if your core implements user-defined "CorExtend" instructions. “CorExtend” is available on cores whose name ends in "Pro".

SB: read-only "SimpleBE" bus mode indicator. If set, means that this core 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 core.

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.

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).

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 24K family cores 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 24K family cores 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 cores 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 core 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 .

2.1.2 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 core. These bits ran out, and other extensions are noted in *Config3*.

Figure 2.2 Fields in the Config1 Register

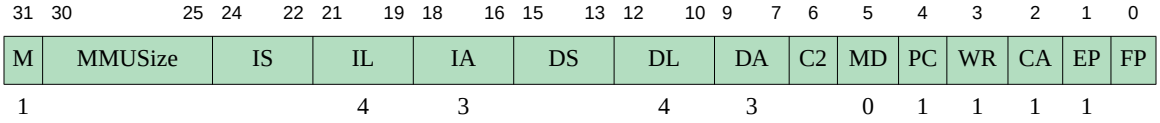
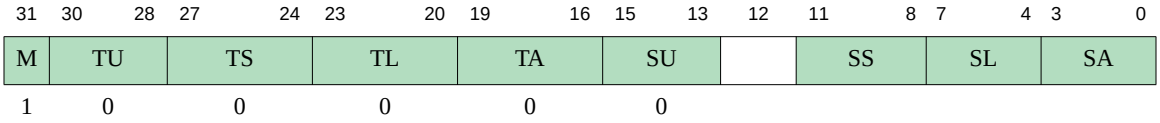


Figure 2.3 Fields in the Config2 Register



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

Config1[MMUSize]: the size of the TLB array (the array has MMUSize+1 entries).

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.

Config1[MD]: 1 if MDMX ASE is implemented in the floating point unit (very unlikely for the 24K core).

Config1[PC]: there is at least one performance counter implemented, see Section 7.4 “Performance counters”.

Config1[WR]: reads 1 because the 24K core always has watchpoint registers, see Section 7.3 “CP0 Watchpoints”.

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

Config1[EP]: reads 1 because an EJTAG debug unit is always provided, see Section 7.1, "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.

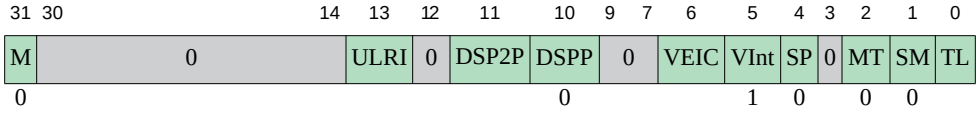
Config2[SU]: implementation-specific bits for secondary cache, if fitted. Can be writable.

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

2.1.3 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 2.4 Fields in the Config3 Register



Fields shown in Figure 2.4 include:

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

Config3[ULRI]: reads 1 if the core implements the *UserLocal* register, typically used by software threads packages. More information in Section B.4.2 “The UserLocal register”.

DSP2P, DSPP: *DSPP* reads 0 because the MIPS DSP extension is not available for this CPU. *DSP2P* distinguishes revision 2 of the DSP ASE .

VEIC: read-only bit from the core 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 5.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)".

VInt: reads 1 to tell you that the 24K core can handle vectored interrupts.

SP: reads 0 to tell you the 24K core does not support sub-4Kbyte page sizes.

MT: reads 0 - no 24K family cores implement the MIPS MT (multithreading) extension.

SM: reads 0, the 24K core does not handle instructions from the "SmartMIPS" ASE.

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

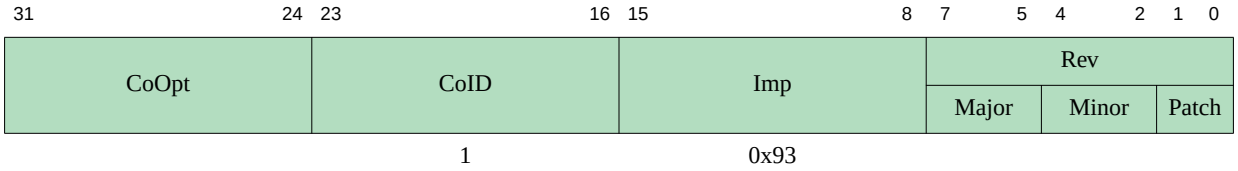
2.1.4 CPU-specific configuration — *Config7*

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 B.4.5 "The Config7 register".

2.2 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 2.5 Fields in the PRId Register



PRId[CoOpt]: Whatever is specified by the SoC builder who synthesizes the core — 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 0x93 for the 24K family. Any processor with different CP0 features must have a new *PRId* field.

PRId[Rev]: The revision number of the core 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. Core licensees can consult [ERRATA] for authoritative information about the revision IDs associated with releases of the 24K core.

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] to hand::

Table 2.2 24K® core releases and *PRId[Revision]* fields

Release Identifier	*PRId[Revision]* Maj.min.patch/hex	Description	Date
2_0_*	2.0.0 / 0x40	General availability of 24K core.	March 19, 2004
3_0_*	3.0.0 / 0x60	COP2 option improvements.	September 30, 2004
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

Table 2.2 24K® core releases and *PRId[Revision]* fields

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

Chapter 3

Memory map, caching, reads, writes and translation

In this chapter:

- Section 3.1, "The memory map": basic memory map of the system.
- Section 3.3, "Reads, writes and synchronization"
- Section 3.4, "Caches"
- Section 3.6, "Scratchpad memory/SPRAM": optional on-chip, high-speed memory (particularly useful when dual-ported to the OCP interface).
- Section 3.7, "The TLB and translation": how translation is done and supporting CP0 registers.

3.1 The memory map

A 24K core system can be configured with either a TLB (virtual memory translation unit) or a fixed memory mapping.

A TLB-equipped CPU 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 3.1. The TLB gives you access to a full 32-bits physical address on the system interface. More information about the TLB in Section 3.7, "The TLB and translation".

Table 3.1 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 2.1. 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.

3.2 Fixed mapping option

To save chip area for applications not needing a full TLB, your core can use a simple fixed mapping (“FMT”) memory translator, which plays the same role. You can find out whether a core has fixed mappings by reading the CP0 field *Config[MT]* (see Figure 2.1 and descriptions). With the fixed mapping option, virtual address ranges are hardwired to particular physical address windows, and cacheability options are set through CP0 register fields as summarized in Table 3.2:

Table 3.2 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.

3.3 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.

3.3.1 Read/write ordering and cache/memory data queues in the 24K® core

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 24K core. 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 core 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 core 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. 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 24K core’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 24K core 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.

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

  The 24K core - 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. 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 3.4.6 “L1 Cache instruction timing”.

3.3.2 The “sync” instruction in 24K® family cores

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`**[4].

But sometimes it’s useful to know more precisely what **`sync`** does on a particular core. On 24K **`sync`**:

4. 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[5]. Not all systems do this. See Section B.4.5 "The Config7 register" for more details.

3.3.3 Write gathering and "write buffer flushing" in 24K® family cores

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 3.4.3 below.

3.4 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 24K core 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 3.4.8, "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 24K core's primary caches are 4-way set associative.

But don't hard-wire any of this information into your software. Instead, probe the *Config1* register defined by [MIPS32] (and described in Section 2.1.2 "The Config1-2 registers") to determine the shape and size of the L1 and any L2 cache.

3.4.1 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 core's standard 64-bit OCP system interface, and when you fit it everything else is attached to the core *through* the L2 cache, which has a system-side interface for that purpose. The core-side interface is enhanced and augmented to support **`cache`** instructions targeted at the L2, and to carry back performance counter information and so on.

5. 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.

- 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 3.4.13 “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 3.4.2 “Cacheability options” for details.

- The L2 cache can run synchronously to the CPU core, 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.

  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.

3.4.2 Cacheability options

Any read or write made by the 24K core 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 3.3 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 3.2.)

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

Table 3.3 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 3.4.3, "Uncached accelerated writes".

3.4.3 Uncached accelerated writes

The 24K core 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 24K core’s 64-bit memory interface.

3.4.4 The cache instruction and software cache management

The 24K core’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 3.4.7 “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 3.1.

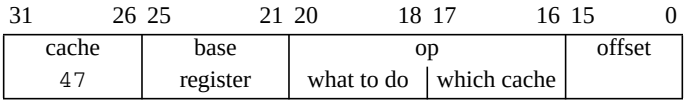


Figure 3.1 Fields in the encoding of a cache instruction

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

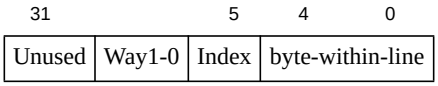
and then adds a 3-bit field which encodes a command to be carried out on the line the instruction selects.

Before we list out the individual commands in Table 3.4; 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 Section 2.1.2 “The Config1-2 registers”) to know exactly where the field boundaries are, but your address is used something like this:



  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].

Table 3.4 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 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 3.1 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 3.6 “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 3.4.8 “Cache aliases” below.
5		*Sorry, different meanings for code “5” on L1 I-cache.*
	Writeback invalidate	On the L1D-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.

3.4.5 Cache instructions and CP0 cache tag/data registers

MIPS Technologies' cores 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 used here and in C header files. I hope Table 3.1 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 3.1 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*

3.4.6 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 3.3 "Reads, writes and synchronization") for potentially matching entries[6]. 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 3.5.

Table 3.5 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

6. In earlier versions of the 24K and 34K family cores, 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.

3.4.7 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.

3.4.8 Cache aliases

The 24K core 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 24K 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[7]...

So the 24K core 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 core 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 24K cores are built to strengthen hit-type I-cache invalidate

7. 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.

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. Refer to Section B.4.5 "The Config7 register" for details.

3.4.9 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[8] - but note that **synci** won't do the job, and should not be used on data/instruction locations which are cache-locked.

3.4.10 Cache initialization and tag/data registers

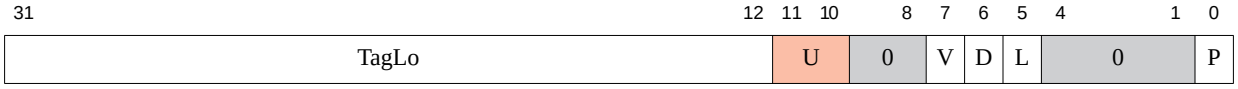
The cache tag/data registers (listed out in Table 3.1) are used for staging tag information being read from or written to the cache (the 24K core has no "TagHi" registers, which are only needed for CPUs with a bigger physical address range). [MIPS32] declares that the contents of these registers is implementation 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 24K core.

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 3.2 shows all the fields:

Figure 3.2 Fields in the *TagLo* Registers



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 3.5 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 24K core but which might not always read zero.

V: 1 when this cache line is valid.

8. 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.

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 3.4.9, "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 **`cacheIndexStoreTag`** 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]*.

3.4.11 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 3.4.13 “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 3.6 “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 B.4.6 “Cache registers in special diagnostic modes”.

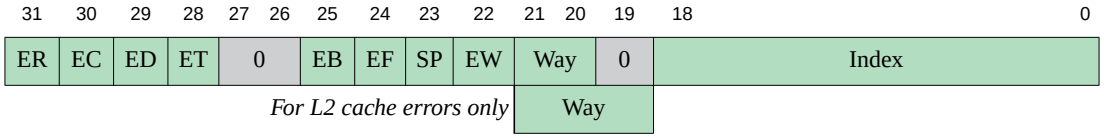
3.4.12 Parity error exception handling and the CacheErr register

The 24K core 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 3.3 Fields in the CacheErr Register



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.

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.

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.

SP: error affecting a scratchpad RAM access, see *Section 3.6, "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*.

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 24K core's largest configurable (4 ways by 16KB) L1 cache option.

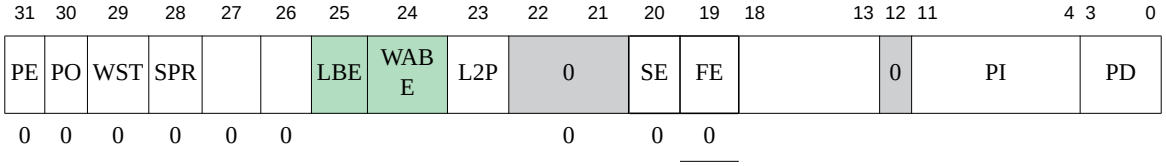
Two other fields are related to the processing of cache errors. Other implementations have laid claim to all of the bits in this register, so these bits were relegated to the *ErrCtl* register. The FE and SE bits in that register are used to detect nested cache errors and are described in the next section.

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 3.1 and its notes above for how to do that.

3.4.13 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 core in the first place).

Figure 3.4 Fields in the ErrCtl Register



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 3.6, "Scratchpad memory/SPRAM".

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

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.

3.5 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 24K core 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..

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 24K core'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.

3.6 Scratchpad memory/SPRAM

The 24K core (like most of MIPS Technologies' cores) 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 24K 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[9].

9. 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.

Probing for SPRAM configuration

The presence of scratchpad RAM in your core is indicated by a “1” bit in one or both of the CP0 *Config[ISP,DSP]* register flags described in Figure 2.1. 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 3.4.13 “ErrCtl register”).

Now a **`cache Index_Load_Tag_D, KSEG0_BASE+0`**[10] 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 3.5.

Figure 3.5 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 cores 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 24K core 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. Note that he 24K core’s instruction memory really is 128 bits wide, so you’ll need two cacheops to fully write a specific index. 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.

3.7 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

10. The instructions are written as if using C “`#define`” names from [m32c0 h]

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.

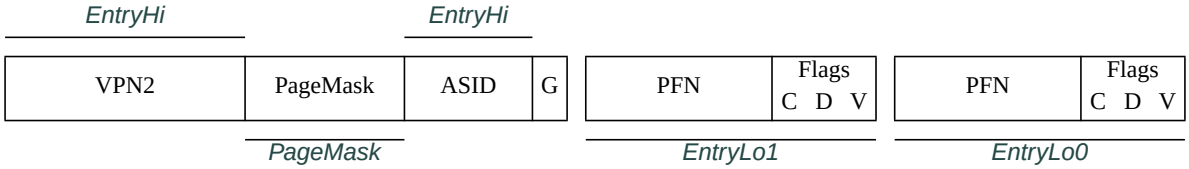
The hardware TLB is relatively small, configurable with 16, 32 or 64 entries (read *Config1[MMUSize]* for the number configured for your core). Each entry can map a 2-page-size virtual region to a pair of physical pages. Entries can map different size pages, too.

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].

3.7.1 A TLB entry

Let’s start with a sketch of a TLB entry. For MIPS32 cores, that consists of a virtual address portion to match against and two output sections, something like Figure 3.6 - which also shows which TLB fields are carried in which CP0 registers.

Figure 3.6 Fields in a 24K® core 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 24K core 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.

3.7.2 Live translation and micro-TLBs

When you're really tuning out the last cycle, you need to know that in the 24K core 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.

3.7.3 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.

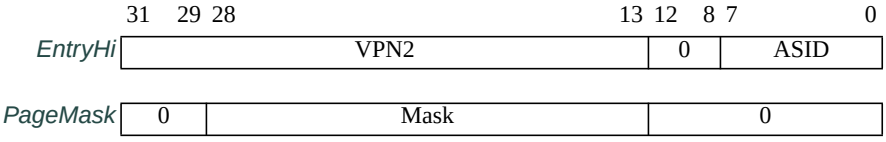
Of these: *Index* determines which TLB entry is accessed by **`tlbwi`**. It's also used for the result of a **`tlbp`** (the instruction you use to see whether a particular address would be successfully translated by the CPU). *Index* only implements enough bits to index the TLB, however big that is; 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.

3.7.4 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 3.7.

Figure 3.7 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.

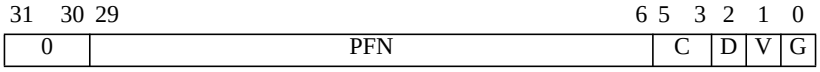
PageMask[Mask]: acts as a kind of backward mask, in that a 1 bit means "don't compare this address bit when matching this address". However, only a restricted range of *PageMask* values are legal (that's with "1"s filling the *PageMask[Mask]* field from low bits upward, two at a time):

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 1Mbyte page sizes; other page sizes are down-converted to 4Kbyte or 1Mbyte 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 3.8.

Figure 3.8 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 24K core 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 3.3 in Section 3.4.2, "Cacheability options" on page 27 .

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.

3.7.5 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 cores protect themselves and you by taking a "machine check" exception if a TLB update would have created a duplicate entry - but in the 24K core that only happens if both entries are valid. Some earlier MIPS Technologies cores 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 24K core, where we repeat that the machine check exception doesn't happen unless the old and new entry are both marked as valid.

3.7.6 TLB exception handlers — *BadVaddr* and *Context*

These two 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 B.4.)

Context contains the useful mix of pre-programmed and borrowed-from-*BadVAddr* bits shown in Figure 3.9.

Figure 3.9 Fields in the Context Register

Context[PTEBase,BadVPN2]: the *PTEBase* field is just software-writable and readable, with no hardware effect.

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.

Chapter 4

Programming the 24K® core 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 4.1, "User-mode accessible “Hardware registers”"
- Section 4.2, "Prefetching data": how it works.
- Section 4.3, "Using “synci” when writing instructions": writing instructions without needing to use privileged cache management instructions.
- Section 4.4, "The multiplier": multiply, multiply/accumulate and divide timings.
- Section 4.5, "Tuning software for the 24K® family pipeline": for determined programmers, and for compiler writers. It includes information about the timing of the DSP ASE instructions.
- Section 4.6 “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.

4.1 User-mode accessible “Hardware registers”

The 24K core 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 5.6 “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[11]; 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.

11. 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 you read a value of "2", then *Count* increments every 2 cycles, at half the pipeline clock rate. For 24K family cores 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 B.4.2 "The UserLocal register" for more details

4.2 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[12].

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 24K family cores 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.

The 24K core acts on hints as summarized in Table 4.1.

4.3 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 4.1, "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 5.1 "Hazard barrier instructions".

12. This isn't quite true any more; **pref** with the "PrepareForStore" hint can zero out some data which wasn't previously zero.

Table 4.1 Hints for “pref” instructions

Hint No	*Hint Name*	*What happens in the 24K core*	*Why would you use it?*
0	load	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.
1	store		
4	load_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
5	store_streamed		
6	load_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.
7	store_retained		
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 24K core 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.

4.4 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*.

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 24K core 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.

4.5 Tuning software for the 24K® 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.

The 24K core is a pipelined design, and the pipeline and some of its consequences are described in Section 1.6 “A brief guide to the 24K® core implementation”. 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.

4.5.1 Cache delays and mitigating their effect

In a typical 24K 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!

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 24K core 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 4.2, "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.

4.5.2 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!

4.6 Tuning floating-point

It seemed to make more sense to put this information into the FPU chapter: read from Section 6.5 “FPU pipeline and instruction timing”.

4.6.1 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 1.6 "A brief guide to the 24K® core implementation" 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.

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[13] 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 24K family. If that's difficult in your environment and the code might need to be portable, it's probably better not to use this.

4.6.2 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[14]. 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 two classes of registers to consider:

- General purpose registers ("GPR");
- The *hi*/*lo* pair ("ACC");

So that gives us two tables.

13. The "likely" in the instruction name is historical, and pretty misleading.
14. 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 4.2 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 an **mthi**/**mtlo** instruction.

Delays caused by "lazy producers" delivering values late

Table 4.3 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**).		
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 4.2.) 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 4.3.)

[3] The **sw** will be 4 clocks late starting while it waits for a result from the multiply pipe (the second box of Table 4.3.)

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 4.3), plus another because that data is required to form the address (first box of Table 4.2.)

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 7.2, "PDtrace™ instruction trace facility").

Chapter 5

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 5.1, "Hazard barrier instructions".
- New interrupt facilities described in Section 5.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)";
- That led on to Section 5.3.1 "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 5.4, "Shadow registers".
- How to get at any power-saving features, in Section 5.5, "Saving Power"
- How to control user-privilege access to "hardware registers", in Section 5.6 "The HWREna register - Control user rdhwr access".

5.1 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 hazard barriers 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**.

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.

5.2 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 core 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 5.4, "Shadow registers". Interrupt handlers using shadow registers avoid the overhead of saving and restoring user GPR values.

- The *Cause[TI]* and *Cause[PCI]* bits (see the notes to Figure B.2) provide a direct indication of pending interrupts from the on-core timer 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 5.1.

Figure 5.1 Fields in the IntCtl Register

IntCtl[IPTI,IPPCI]: *IPTI* and *IPPCI* are read-only 3-bit fields, telling you how internal timer and performance counter 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]* and *IntCtl[IPPCI]* are between 2 and 7.

The timer and performance counter interrupt signals are taken out to the core interface and the SoC designer connects them back to one of the core's interrupt inputs. The SoC designer is supposed to hard-wire some core inputs which show up as the *IntCtl[IPTI,IPPCI]* fields to match.

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.

5.2.1 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 5.1 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.

5.2.2 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 core has the vectored-interrupts feature - but all cores in the 24K 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".

5.2.3 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[15]: 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 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 B.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[16].

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 core, and in most cases will become inputs - presumably low-priority ones - to the EIC-compliant interrupt controller.

15. 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).
16. 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.

In EIC mode the usual association of the internal timer and performance-counter overflow interrupts with individual bits of *Cause[IP]* is lost. Timer and performance counter interrupts are turned into output signals from the core, and will themselves become inputs to the interrupt controller. Ask your system integrator how they are wired.

5.3 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[17], 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.

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 B.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) have their exception entry points in kseg0. By setting *EBase* differently on each CPU, you can give them distinct exception handlers.

Figure 5.2 Fields in the EBase Register

17. Even this address can be changed by a brave and determined SoC integrator, see the note on RBASE in Section 5.3.1 "Summary of exception entry points".

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 core interface bus *SI_CPUNum*, which the SoC designer will tie to some suitable value).

5.3.1 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 5.1. 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, 24K family cores can be configured to use a different base address by fixing some input signals to the core. Specifically, if the core 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.

Table 5.1 All Exception entry points

Memory region	*Entry point*	*Exceptions handled here*
EJTAG probe-mapped	`0xFF20.0200`	EJTAG debug, when mapped to “probe” memory.
ROM-only entry points	`RBASE+0x0480`	EJTAG debug, when using normal ROM memory.
	`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 5.2, "MIPS32® Architecture Release 2 - enhanced interrupt system(s)".
	`BASE+0x180`	All others

5.4 Shadow registers

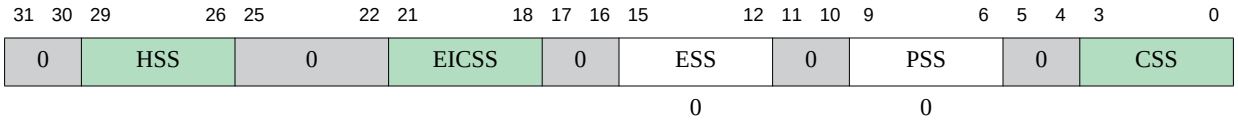
In hardware terms, shadow registers are deceptively simple: just add one or more extra copies of the register file. If you can automatically change register set on an exception, the exception handler will run with its own context, and without the overhead of saving and restoring the register values belonging 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`**.

Selecting shadow sets - SRSCtl

The shadow set selectors are in the *SRSCtl* register, shown in Figure 5.3.

Figure 5.3 Fields in the SRSCtl Register



SRSCtl[HSS]: the highest-numbered register set available on this 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.

This field is read-only.

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 5.2.3 "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 core 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:

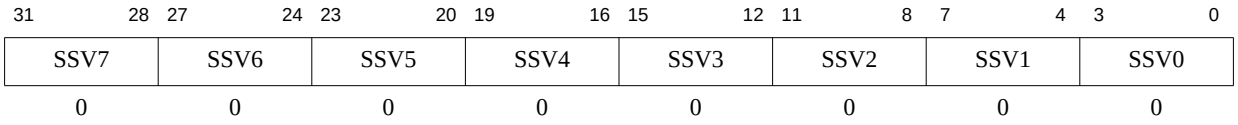
- Exceptions occurring with *Status[EXL]* already set;
- Cache error exceptions, where the return address is loaded into *ErrorEPC*;
- EJTAG debug exceptions, where the return address is loaded into *DEPC*.

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 core 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 5.4.

Figure 5.4 Fields in the SRSMap Register



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.

5.5 Saving Power

There are basically just a couple of facilities:

- The **`wait`** instruction causes the CPU to enter a low-power sleep mode until woken by an interrupt. Most of the core logic is stopped, but the *Count* register, in particular, continues to run.

  In some cores — 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. 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 B.4.1 "Status register", and Section B.4.5 "The Config7 register".

- The *Status[RP]* bit: this doesn't do anything inside the core, but its state is made available at the core interface as *SI_RP*. Logic outside the core 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.

5.6 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 4.1 "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 5.5 Fields in the HWREna Register

HWREna[Impl]: Read 0. If there were any implementation-dependent hardware registers, you could control access to them here. Currently, no 24K family core 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 6

Floating point unit

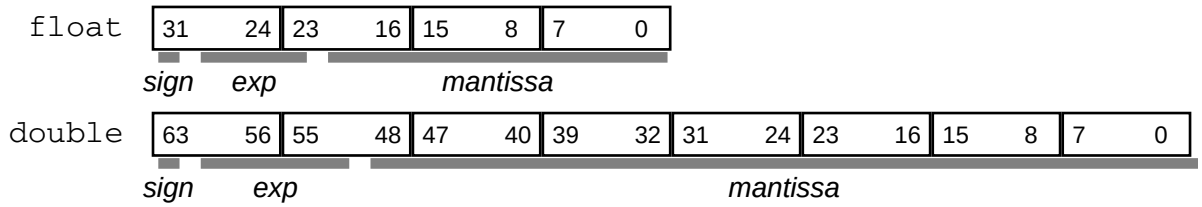
The 24Kf™ member of the 24K family has a hardware floating point unit (FPU). This:

- *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 24K family integer core 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 C.3 “FPU changes in Release 2 of the MIPS32® Architecture” or for full details [MIPS32].
- *Usually runs at half the integer core’s clock rate*: the design is tested to work with the FPU running at the core speed, but in likely processes the FPU will then limit the achievable frequency of the whole core. 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 24K core 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 6.4.2, "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 6.5, "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.

6.1 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 6.1) what 32-bit and 64-bit floating point numbers on MIPS architecture CPUs look like.

Figure 6.1 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 6.1 will be at the lowest memory location when the core is configured big-endian, and the highest memory location when the core is little-endian.

6.2 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.

6.3 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.

6.4 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.

6.4.1 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 6.1 below.

It's overwhelmingly popular to keep *FENR* zero and thus never generate an IEEE exception; see Section 6.5, "FPU pipeline and instruction timing" for why this is a particularly good idea if you want the best performance.

6.4.2 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 24K core'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[18].

18. 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 6.1, "FPU (co-processor 1) control registers"), or provide a software emulator and resign yourself to a small number of “unimplemented” exceptions.

6.4.3 FPU control register maps

There are five FP control registers:

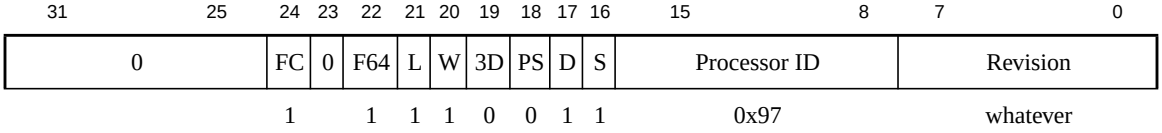
Table 6.1 FPU (co-processor 1) control registers

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	25	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.
FEXR	26	
FENR	28	

The FP implementation (FIR) register

Figure 6.2 shows the fields in *FIR* and the read-only values they always have for 24K family FPUs:

Figure 6.2 Fields in the FIR register



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 6.3 shows all these registers and their bits

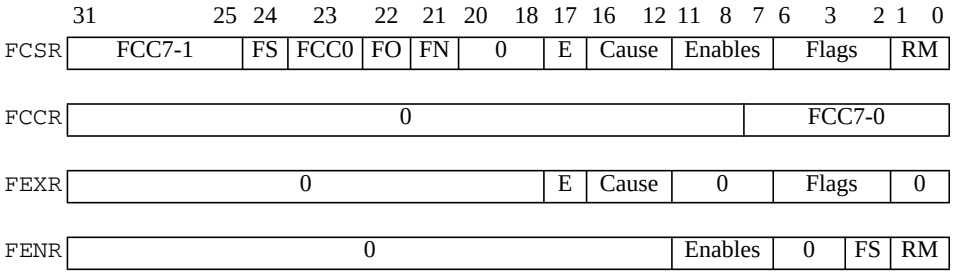


Figure 6.3 Floating point control/status register and alternate views

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[19]. 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 6.4.2, "FPU “unimplemented” exceptions (and how to avoid them)".

19. 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[20] 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*

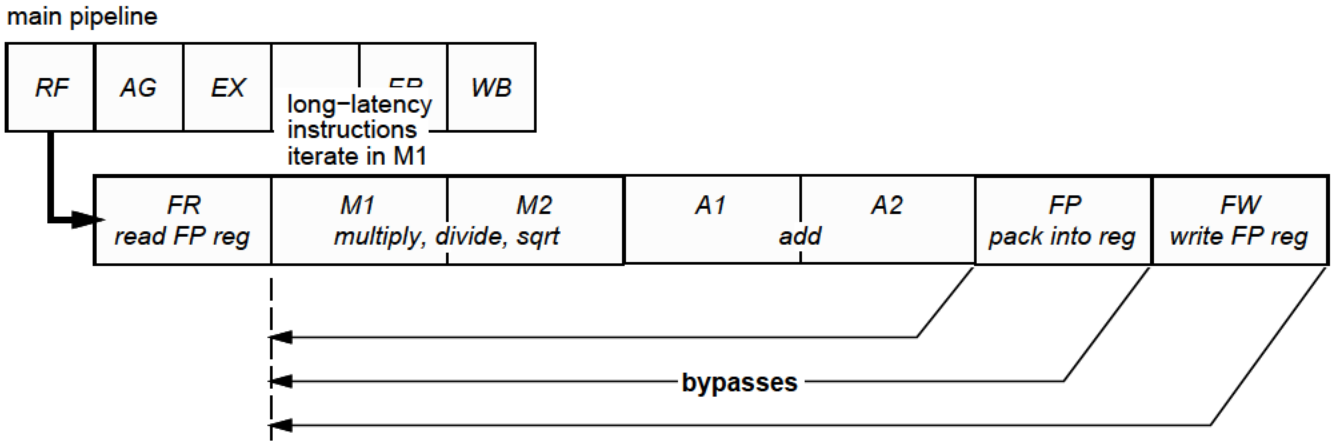
6.5 FPU pipeline and instruction timing

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 core - 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 24K family cores, the FPU will commonly be built with a half rate clock. You can find how your core is set up by looking at the *Config7[FPR1-0]* bits, defined in the notes to Figure B-3.

Nonetheless, this is a powerful 64-bit floating point unit which can deliver very good performance. The FPU pipeline is shown in Figure 6.4.

20. 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 6.4 Overview of the FPU pipeline



FPU instructions are fetched and despatched through the integer core. All instructions (including FP instructions) proceed through the core 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 core, instructions issued from the integer core 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.

6.5.1 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.

6.5.2 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 6.2. 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 6.2 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

6.5.3 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.

6.5.4 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 6.1, "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.

6.5.5 Delays when main pipeline waits for FPU to accept an instruction

An FPU running slower than the core 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.

6.5.6 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.

6.5.7 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.

6.5.8 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 7

24K® core features for debug and profiling

In this chapter you'll find:

- Section 7.1, "EJTAG on-chip debug unit"
- Section 7.2 "PDtrace™ instruction trace facility"
- Section 7.3 "CP0 Watchpoints" - monitor code and data access without using EJTAG.
- Section 7.4 "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]).

7.1 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[21].

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 7.1.5, "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 7.1.2, "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";

21. 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 cores tend to have fewer options and to implement the bulk of the EJTAG specification. The 24K core 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.

7.1.1 Debug communications through JTAG

The chip's JTAG pins give an external probe access to a special registers inside the core. 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 7.1.

Table 7.1 JTAG instructions for the EJTAG unit

JTAG "Instruction"	*Description*
IDCODE	Reads out the MIPS core and revision - not very interesting for software, not described further here.
ImpCode	Reads bit-field showing what EJTAG options are implemented - see Figure 7.5 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 7.1.5, "The "dseg" memory decode region".
EJTAG_CONTROL	Package of flags and control fields for the probe to read and write; see Figure 7.6 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 7.6. 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*.
TCBCONTROLA TCBCONTROLB TCBCONTROLC TCBCONTROLD TCBCONTROLE	Access registers used to control "PDtrace" instruction trace output, if available. See Section 7.2.1 "24K core-specific fields in PDtrace™ JTAG-accessible registers" - only the core-specific fields in these registers are documented here.
PCSAMPLE	Access register which holds PC sample value, see Section 7.1.12, "PC Sampling with EJTAG".

7.1.2 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 7.1 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 7.1.2, "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 - see Table 5.1.

7.1.3 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.

7.1.4 Single-stepping

When control returns from debug mode with a **`deret`** and the single-step bit *Debug[SSt]* is set, the instruction selected by *DEPC* will be executed in non-debug context[22]; 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.

7.1.5 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 7.2.

22. If *DEPC* points to a branch instruction, both the branch and branch-delay instruction will be executed normally.

Table 7.2 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
0xFF30.1000			IBS register	0xFF30.1000
			I-breakpoint #1 regs	
0xFF30.1100			IBA1	0xFF30.1100
0xFF30.1108			IBM1	0xFF30.1108
0xFF30.1110			IBASID1	0xFF30.1110
0xFF30.1118			IBC1	0xFF30.1118
			I-breakpoint #2 regs	
0xFF30.1200			IBA2	0xFF30.1200
0xFF30.1208			IBM2	0xFF30.1208
0xFF30.1210			IBASID2	0xFF30.1210
0xFF30.1218			IBC2	0xFF30.1218
			same for next two	
			...	
0xFF30.2000			DBS register	0xFF30.2000
			D-breakpoint #1 regs	
0xFF30.2100			DBA1	0xFF30.2100
0xFF30.2108			DBM1	0xFF30.2108
0xFF30.2110			DBASID1	0xFF30.2110
0xFF30.2118			DBC1	0xFF30.2118
0xFF30.2120			DBV1	0xFF30.2120
0xFF30.2124			DBVHi1	0xFF30.2124
			D-breakpoint #2 regs	
0xFF30.2200			DBA2	0xFF30.2200
0xFF30.2208			DBM2	0xFF30.2208
0xFF30.2210			DBASID2	0xFF30.2210
0xFF30.2218			DBC2	0xFF30.2218
0xFF30.2220			DBV2	0xFF30.2220
0xFF30.2224			DBVHi2	0xFF30.2224
0xFF30.2228				0xFF30.2228
0xFFFF.FFFF				0xFFFF.FFFF

Notes on Table 7.2:

- *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 7.1).

- *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 core. 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.

7.1.6 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:

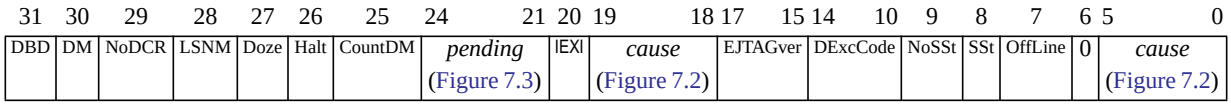
- *Debug* which has configuration and control bits, and is detailed below;

- *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 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 7.2, information about regular exceptions which want to happen but can’t because you’re in debug mode in Figure 7.3, and everything else. The "everything else" category includes the most important fields and comes first, in Figure 7.1.

Figure 7.1 Fields in the EJTAG CP0 Debug register



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 cores 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 24K core, 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 7.3 below.

EJTAGver: read-only - tells you which revision of the specification this implementation conforms to. On the 24K core it reads `3` for version 3.1. 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

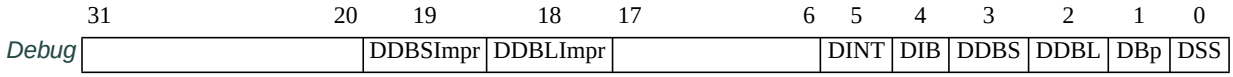
DExcCode: Cause of any non-debug exception you just handled from within debug mode - following first entry to debug mode, this field is undefined. The value will be one of those defined for *Cause[ExcCode]*, as shown in Table B.4.

NoSSt: read-only - reads 0 because single-step is implemented (it always is on MIPS Technologies cores).

SSt: set 1 to enable single-step.

OffLine: a feature used in multithreading CPUs.

Figure 7.2 Exception cause bits in the debug register



DDBSImpr: imprecise store breakpoint - see Section 7.1.11, "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`**.

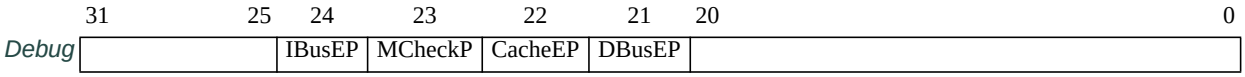
DDBS: precise store breakpoint.

DDBL: precise load breakpoint.

DBp: any sort of match with a hardware breakpoint.

DSS: single-step exception.

Figure 7.3 Debug register - exception-pending flags



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 24K core, so this can’t happen and the field is always zero.

MCheckP: machine check pending (usually an illegal TLB update). As above, the machine check is always precise on the 24K core, so this is always zero.

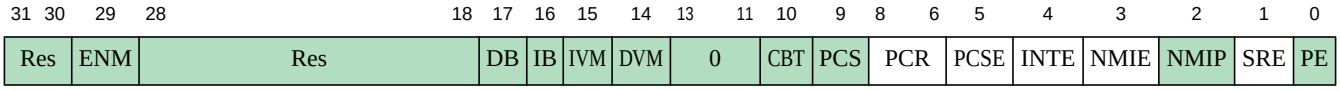
CacheEP: cache parity error pending.

DBusEP: bus error on data access pending.

7.1.7 The DCR (debug control) memory-mapped register

This is the only memory-mapped EJTAG register apart from the hardware breakpoints (described in the next section). It’s found in “drseg” at location 0xFF30.0000 as shown in Table 7.2 (but only accessible if the CPU is in debug mode). The fields are in Figure 7.4:

Figure 7.4 Fields in the memory-mapped DCR (debug control) register



Where:

ENM: (read only) reports CPU endianness (1 == big).

DB/IB: (read only) 1 if data/instruction hardware breakpoints are available, respectively. The 24K core 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.

CBT: (read-only) tells you if a complex breakpoint block is implemented.

PCS, PCR: *PCS, PCSE* reads 1 if the PC sampling feature is available, as it can be on the 24K core. Then *PCR* is a three-bit field defining the sampling frequency as one sample every $2^{(5+PCR)}$ cycles. See Section 7.1.12, "PC Sampling with EJTAG" for details.

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 24K core 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 7.6.

7.1.8 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 core by the SoC team, not very interesting) and *ImpCode* (Figure 7.5); then there's the main CPU interaction control/status register *EJTAG_CONTROL* (Figure 7.6).

Figure 7.5 Fields in the JTAG-accessible ImpCode register

31 29	28 25	24	23 21	20 17	16	15	14	13 1	0
EJTAGver	0	DINTsup	ASIDsize	0	MIPS16	0	NoDMA	0	MIPS32/64
2 = 2.6		*see note*	*see note*		1		1		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 core 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 core has been built with the no-TLB option (i.e. a fixed-mapping MMU).

MIPS16: 1 because the 24K core always supports the MIPS16 instruction set extension.

NoDMA: 1 - MIPS Technologies cores do not provide EJTAG "DMA" (which would allow a probe to directly read and write anything attached to the 24K core'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 7.6:

Figure 7.6 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:

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 EJTAG_ADDRESS[1-0]	*Size code* EJTAG_CONTROL[Psz]	*Transfer Size*
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 core, which can be used to reset non-core logic (ask your SoC designer whether it's connected to anything).

For this and all other fields which change core 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 core, 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[23] (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]*.

7.1.9 EJTAG breakpoint registers

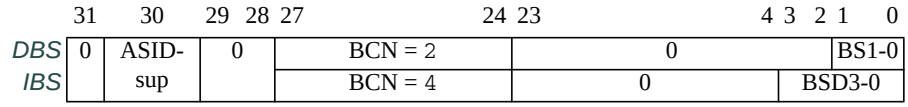
It's optional whether the 24K core has EJTAG breakpoint registers. But if it has instruction breakpoints, it has four of them; and if it has data breakpoints, it has two. The breakpoints:

- 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 24K core - see Section 7.1.11, "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 7.2, "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 cores with hardware breakpoints always include the value check, so these bits read zero anyway. So the registers are as shown in Figure 7.7.

23. The ROM-exception-area debug entry point can be relocated by hardware option, see Table 5.1 and its notes.

Figure 7.7 Fields in the IBS/DBS (EJTAG breakpoint status) registers



Where:

ASIDsup: is 1 if the breakpoints can use ASID matching to distinguish addresses from different address spaces; on the 24K core that's available if and only if a TLB is fitted.

BCN: the number of hardware breakpoints available (two data, 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).

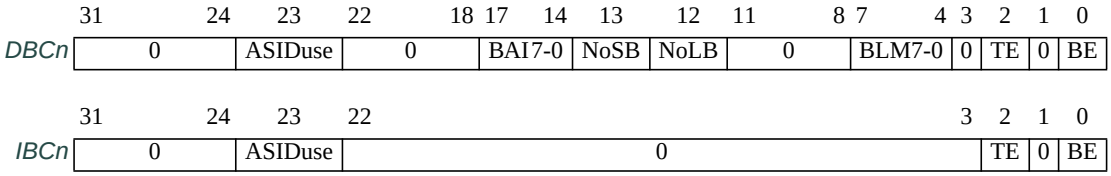
  The higher 24 bits of each of these registers is always zero.
- *DBVn, DBVHin* the value to be matched on load/store breakpoints. *DBCHin* defines bits 63-32 to be matched for 64-bit load/stores: the 32-bit[24] 24K has 64-bit load/store instructions for floating point.

  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 7.8.

24. A JTAG hardware breakpoint for a real 64-bit CPU would have 64-bit *DBVn* registers, so wouldn't need *DBVHin*.

Figure 7.8 Fields in the hardware breakpoint control registers (IBCn, DBCn)



The fields are:

ASIDuse: set 1 to compare the ASID as well as the address.

BAI7-0: "byte (lane) access ignore"[25] - 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*[26] 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 7.2 "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!

7.1.10 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:

    1. The breakpoint control register enable bit *IBAn[BE]*/*DBAn[BE]* is set.

25. Why are there 8 bytes, when the 24K core 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 24K core has a 64-bit data bus (and in fact can do 64-bit load and store operations, for example for floating point values).
26. "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".

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.

7.1.11 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 24K core 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.

7.1.12 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 to your core, it runs continuously. It doesn't even stop when the CPU is hanging on a **`wait`** instruction (time spent waiting is still time you might want to measure). You can choose to sample as often as once per 32 cycles or as rarely as once per 4096 cycles[27]; 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.

The hardware stores not only 32 bits of the instruction address, but also the then-current ASID (so you can interpret the virtual PC) and an always-written-1 "new" bit which a probe can use to avoid double-counting the same sample.

27. Since it runs continuously, it's a good thing that from reset the sampling period defaults to its maximum.

7.2 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.

24K family cores 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 24K core 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 24K core can be configured with up to 8 “trace triggers”, allowing you to start and stop tracing based on EJTAG breakpoint matches: see Section 7.1.9, "EJTAG breakpoint registers" above and Table 7.15 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 core, 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 core. 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.

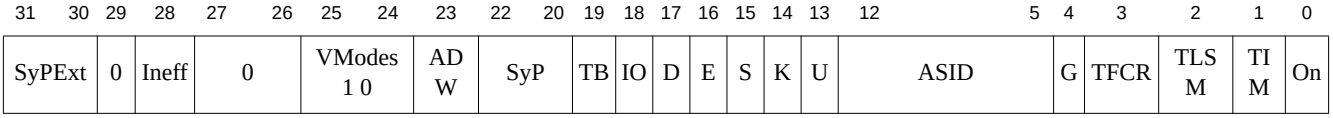
In practice the PDtrace logic depends on the existence of an EJTAG unit (described in the previous section) and an enhanced EJTAG probe. To benefit from on-probe trace memory, the probe will need to attach to PDtrace-specific signals.

This manual describes only the lowest-level building blocks as visible to software. For real hardware information refer to [PDTRACETCB]; for 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.

7.2.1 24K core-specific fields in PDtrace™ JTAG-accessible registers

The PDtrace system is controlled by the JTAG-accessible registers TCBCONTROLA, TCBCONTROLB, TCBCONTROLC, . Normally they are not visible to software running on the CPU, but we’ll document fields and configured values which are specific to 24K family CPUs.

Figure 7.9 Fields in the TCBCONTROLA register



31	30	29	28	27	26	25	24	23	22	20	19	18	17	16	15	14	13	12	5	4	3	2	1	0
SyPExt		0	Ineff	0		VModes 1 0		AD W	SyP		TB	IO	D	E	S	K	U	ASID		G	TFCR	TLS M	TI M	On

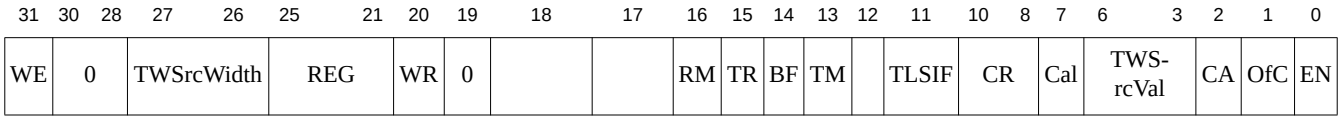
In TCBCONTROLA:

VModes: reads “1 0”, showing that 24K family cores 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 core-specific-inefficieny tracing is enabled.

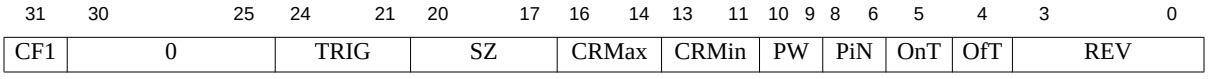
Figure 7.10 Fields in the TCBCONTROLB register



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			RM	TR	BF	TM		TLSIF	CR		Cal	TWS-rcVal		CA	OfC	EN

In TCBCONTROLB:

Figure 7.11 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 24K core is a single-issue (single pipeline) processor.

REV: reads 1, denoting compliance with revision 4.0 of the TCB specification.

7.2.2 CP0 registers for the PDtrace™ logic

There are three categories of registers:

- *TraceControl* , *TraceControl2 (*Figure 7-12/Figure 7-13 *)*: allow the software to take charge of what is being traced.
- *UserTraceData*(Section 7.2.4 “UserTraceData 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 7.15): controls whether and how individual EJTAG breakpoint trace triggers take effect.

Figure 7-12 Fields in the TraceControl Register

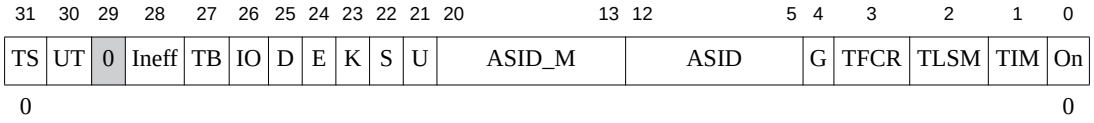


Figure 7-13 Fields in the TraceControl2 Register

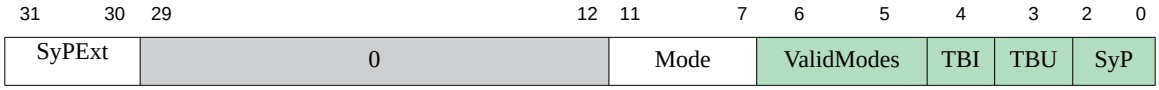


Figure 7.14

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.*Ineff*: set to 1 to indicate that core-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.

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[28]. 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.

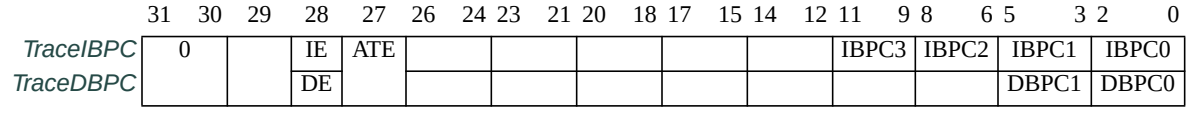
TBI	*TBU*	*On-chip or probe trace memory?*
0	0	only on-chip memory available
0	1	only probe memory available
1	0	Both available, currently using on-chip
1	1	Both available, currently using probe

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}.

7.2.3 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 7.15

Figure 7.15 Fields in the TraceIBPC/TraceDBPC registers



In either *TraceIBPC* or *TraceDBPC*:

IE,DE: master 1-to-enable bit for triggers from EJTAG instruction and data breakpoints respectively.

28. Prior to v4 of the PDtrace specification, this field was in *TraceControl*, and was too small to allow all conditions to be specified independently.

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 24K core.

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 24K core 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).

7.2.4 UserTraceData 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). 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.

7.2.5 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 core-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”.

7.3 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 24K cores you get two I-side and two D-side registers.

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.

7.3.1 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.

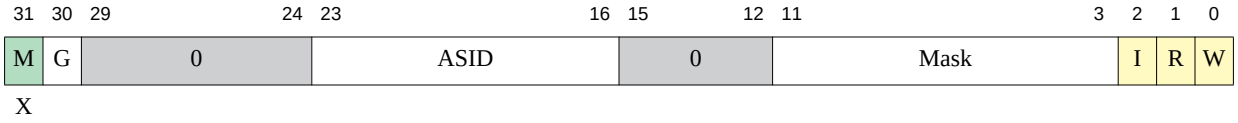
Figure 7.16 Fields in the WatchLo0-3 Register

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). 24K 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.

7.3.2 The WatchHi0-3 registers

Figure 7.17 Fields in the WatchHi0-3 Register



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.

7.4 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.

24K family CPUs are fitted with four 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 7.18.

Figure 7.18 Fields in the PerfCtl Registers

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.

Event: determines which event this counter will count; see Table 7.4 below. Note that the odd-numbered and even-numbered counters mostly count different events, though some particularly important events can use any of the four 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 signals (usually *SI_PCI*) out of the core, which the system integrator will have sent back in, each as one of the core's interrupt inputs.

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 24K core are in Table 7.4. Blank fields are reserved. But before you get there, take a look at the next sub-section...

7.4.1 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 7.4 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.
- *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 Figure 1.1 "Pipeline differences between the 24K® and 4K™ core families".
- *LDQ, FSB, WBB*: CPU queues, described in Section 3.3.1 "Read/write ordering and cache/memory data queues in the 24K® core".
- *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:
  - 24K 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.
- *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 7.4 Performance counter events

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
0	Cycles			P
1	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

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
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. 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
12-13	reserved			
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		T
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

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
25		P	Cycles when main pipeline stops because an ALU operation is taking more than one clock	P
26				
27				
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				
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	reserved			
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	L2 I-miss stall cycles	P	L2 data miss stall cycles	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
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		
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	Branch mispredict: lost cycles resulting from a mispredict (24K and 24KE only).	Pi
47	Relax bubbles	V		

Event No	Counter 0 and 2	Type	Counter 1 and /3	Type
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 3.3.1 "Read/write ordering and cache/memory data queues in the 24K® core".			
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

Appendix A

References

24K™ core family manuals

[SUM]:“MIPS32® 24K® Processor Core Family Software User’s Manual”, MIPS Technologies document MD00343.

[ERRATA]:“MIPS32® 24K® Processor Core Family RTL Errata Sheet”, MIPS Technologies document MD00345. Available only to core licensees or by arrangement.

[INTGUIDE]:““MIPS32® 24K® Processor Core Family Integrator's Guide”, MIPS Technologies document MD00344, available to core licensees, describes the options available with the core.

[24KC_DATA]:“MIPS32 24Kc™ Datasheet”, MIPS Technologies document MD00346.

[24KF_DATA]:“MIPS32 24Kf™ Datasheet”, MIPS Technologies document MD00354.

Other Manuals from MIPS Technologies

First of all, basic architectural documents.

[MIPS32]: The MIPS32 architecture definition cines 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 24K 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:

[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.

Appendix B

CP0 register summary and reference

This appendix lists all the CP0 registers of the 24K 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 24K 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.

B.1 CP0 registers by name

Table B.1 Register Index by Name

Name	Number	Name	Number	Name	Number	Name	Number
BadVAddr	8.0	*DTagLo*	28.2	*L23DataHi*	29.5	*SRSCtl*	12.2
CacheErr	27.0	*EBase*	15.1	*L23DataLo*	28.5	*SRSMap*	12.3
Cause	13.0	*EntryHi*	10.0	*L23TagLo*	28.4	*Status*	12.0
Compare	11.0	*EntryLo0-1*	2.0	*PageMask*	5.0	*TraceControl*	23.1
Config	16.0		3.0	*PerfCnt0-3*	25.1	*TraceControl2*	23.2
Config1-2	16.1-2	*EPC*	14.0		25.3	*TraceDBPC*	23.5
Config3	16.3	*ErrCtl*	26.0		25.5	*TraceIBPC*	23.4
Config7	16.7	*ErrorEPC*	30.0		25.7	*UserLocal*	4.2
Context	4.0	*HWREna*	7.0	*PerfCtl0-3*	25.0	*UserTraceData*	23.3
Count	9.0	*Index*	0.0		25.2	*WatchHi0-3*	19.0-3
Debug	23.0	*IntCtl*	12.1		25.4	*WatchLo0-3*	18.0-3
DDataLo	28.3	*IDataHi*	29.1		25.6	*Wired*	6.0
DEPC	24.0	*IDataLo*	28.1	*PRId*	15.0		
DESAVE	31.0	*ITagLo*	28.0	*Random*	1.0		

B.2 CP0 registers by number

Table B.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
0.0	*Index*	Index into the TLB array	3.7.3, p.40
1.0	*Random*	Randomly generated index into the TLB array	3.7.3, p.40
2.0	*EntryLo0*	Output (physical) side of TLB entry for even-numbered virtual pages	Figure 3.8 , p. 41
3.0	*EntryLo1*	Output (physical) side of TLB entry for odd-numbered virtual pages	Figure 3.8 , p. 41
4.0	*Context*	Mixture of pre-programmed and *BadVAddr* bits which can act as an OS page table pointer.	Figure 3.9 , p. 42
4.2	*UserLocal*	Kernel-writable but user-readable software-defined thread ID	B.4.2, p.105
5.0	*PageMask*	Control for variable page size in TLB entries	Figure 3.7 , p. 40
6.0	*Wired*	Controls the number of fixed (“wired”) TLB entries	3.7.3 , p. 40
7.0	*HWREna*	Select which hardware registers are readable using the **`rdhwr`** instruction in user mode.	Figure 5.5 , p. 59
8.0	*BadVAddr*	Reports the address for the most recent TLB-related exception	3.7.6, p.42
9.0	*Count*	Free-running counter at pipeline or sub-multiple speed	B.4.4, p.107
10.0	*EntryHi*	High-order portion of the TLB entry	Figure 3.7 , p. 40

Table B.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
11.0	*Compare*	Timer interrupt control	B.4.4, p.107
12.0	*Status*	Processor status and control	Figure B.1 , p. 104
12.1	*IntCtl*	Setup for interrupt vector and interrupt priority features.	Figure 5.1 , p. 53
12.2	*SRSCtl*	Shadow register set selectors	Figure 5.3 , p. 57
12.3	*SRSMap*	In VI (vectored interrupt) mode, determines which shadow set is used for each interrupt source.	Figure 5.4 , p. 58
13.0	*Cause*	Cause of last general exception	Figure B.2 , p. 106
14.0	*EPC*	Restart address from exception (no subfields, not described further in this manual)	[MIPS32]
15.0	*PRId*	Processor identification and revision	Figure 2.5 , p. 21
15.1	*EBase*	Exception entry point base address and CPU ID	Figure 5.2 , p. 55
16.0	*Config*	Configuration register	Figure 2.1 , p. 17
16.1-2	*Config1-2*	Configuration for MMU, caches etc	Figure 2.2 , p. 19
16.3	*Config3*	Interrupt and ASE capabilities	Figure 2.4 , p. 20
16.7	*Config7*	24K family-specific configuration	Figure B-3 , p. 108
18.0-3	*WatchLo0-3*	Watchpoint address: *WatchLo0-1* are I-side, and *WatchLo2-3* are D-side	Figure 7.16 , p. 90
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 7.1 , p. 75
23.1	*TraceControl*	Control fields for the PDTrace unit.	Figure 7-12 , p. 86
23.2	*TraceControl2*		
23.3	*UserTraceData*	Software-generated PDTrace information register	7.2.4 , p. 88
23.4	*TraceBPC*	Additional controls for PDTrace start/stop	Figure 7.15 , p. 87
24.0	*DEPC*	Restart address from last EJTAG debug exception	7.1.6 , p. 74
25.0 25.2 25.4 25.6	*PerfCtl0-3*	Performance counter control	Figure 7.18 , p. 91
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 3.4 , p. 36
27.0	*CacheErr*	Cache parity exception control and status	3.4.12, p.34

Table B.2 Cross-referenced list of CP0 registers by number

Nos	Register	Description	Refer to
28.0	*ITagLo*	Cache tag read/write interface for I-, D- and L2 (secondary) cache respectively	3.2, p.33 B.4.6, p.109
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	3.4.12, p.34
31.0	*DESAVE*	Scratch read/write register for EJTAG debug exception handler	7.1.6, p.74

B.3 CP0 registers by function

Table B.3 CP0 registers grouped by function

Basic modes	*Status*	12.0	**Cache Management**	*DDataLo*	28.3	**Profiling**	*PerfCnt0-3*	25.1
Exception control	*Cause*	13.0		*DTagLo*	28.2			25.3
	EPC	14.0		*ErrCtl*	26.0			25.5
OS/userland thread ID	*UserLocal*	4.2		*ErrorEPC*	30.0			25.7
				IDataHi	29.1		*PerfCtl0-3*	25.0
Timer	*Compare*	11.0		*IDataLo*	28.1			25.2
	Count	9.0		*ITagLo*	28.0			25.4
CPU Configuration	*Config*	16.0		*L23DataHi*	29.5			25.6
	Config1-2	16.1-2		*L23DataLo*	28.5	**debug/ analysis**	*WatchHi0-3*	19.0-3
	Config3	16.3		*L23TagLo*	28.4		*WatchLo0-3*	18.0-3
	Config7	16.7	**EJTAG debug**	*DEPC*	24.0	**Control user rdhwr access**	*HWREna*	7.0
	EBase	15.1		*DESAVE*	31.0			
	IntCtl	12.1		*Debug*	23.0	**Parity/ECC control**	*CacheErr*	27.0
	PRId	15.0	**PDtrace block**	*TraceControl*	23.1			
	SRSCtl	12.2		*TraceControl2*	23.2			
	SRSMap	12.3		*TraceDBPC*	23.5			
TLB Management	*BadVAddr*	8.0		*TraceIBPC*	23.4			
	Context	4.0		*UserTraceData*	23.3			
	EntryHi	10.0						
	EntryLo0-1	2.0						
		3.0						
	Index	0.0						
	PageMask	5.0						
	Random	1.0						
	Wired	6.0						

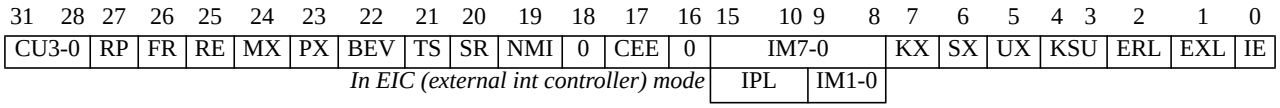
B.4 Miscellaneous CP0 register descriptions

Many CP0 registers in the 24K 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.

B.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 B.1. All fields are writable unless noted otherwise.

Figure B.1 All Status register fields



The 24K 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. Writable when such a coprocessor exists. Since no 24K 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 24K core, but the state of the RP bit is available on the external core interface as the *SI_RP* signal. The 24K 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 24K 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 24K 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 24K CPU).

BEV: "boot exception vectors" - when 1, relocates all exception vectors to near the reset-time start address. See Section 5.3.1 "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.

SR: MIPS32 architecture "soft reset" bit: the 24K 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 5.2.3 "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 24K core is a 32-bit CPU, so these are always zero.

KSU: execution privilege level - basically user or kernel:

0 kernel
1 supervisor - not available on 24K cores
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[29].

IE: global interrupt enable, 0 to disable all interrupts.

B.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.

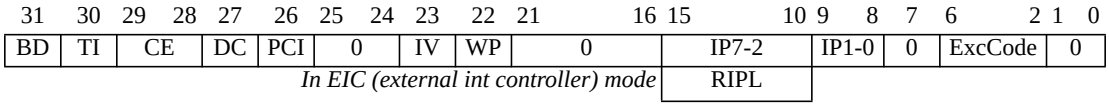
UserLocal was first implemented after the first release of the 24K family of cores. Kernels should check whether this register is implemented by inspecting *Config3[ULRI]*, as described in Section 2.1.3 "The Config3 register". Use of **`rdhwr $29`** will cause an exception in CPUs not implementing this register, providing an opportunity for an OS kernel to simulate it.

29. 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.

B.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 B.2 Fields in the Cause register



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 7.4 “Performance counters”.

IV: (writable) set 1 to use a special exception entry point for interrupts, see Section 5.3.1 “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 5.2 “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 5.2.3 “External Interrupt Controller (EIC) mode”.

ExcCode: what caused that last exception. Lots of values, listed in Table B.4.

Table B.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" - tried to write conflicting TLB entries
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 7.1 "EJTAG on-chip debug unit", and in particular the notes on the *Debug* register in Figure 7.1.
31	–	Reserved

B.4.4 Count and Compare

These two 32-bit registers form a useful and flexible timer. *Count* just counts. For the 24K 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 4.1 "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:

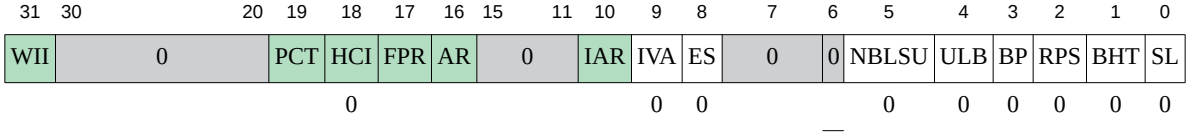
- For a periodic interrupt, simply advance *Compare* by a fixed amount each time (and check for the possibility that *Count* has overrun it).
- To set a timer for some point in the future, just set *Compare* to an increment more than the current value of *Count*.

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. Your system integrator should wire information pins to show where the timer interrupt is connected: see the notes on *IntCtl[IPTI]* below Figure 5.1. However, if you have an "EIC" interrupt controller (see Section 5.2.3 "External Interrupt Controller (EIC) mode") you'll need to send the timer interrupt all the way out to the interrupt controller and back.

B.4.5 The Config7 register

The *Config7* register holds CPU-specific one-time setup and basic information fields.

Figure B-3 Fields in the Config7 Register



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 will always be enough to terminate a **`wait`** instruction.

24K 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]*. 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 7.4 "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 3.4.8 "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.

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.

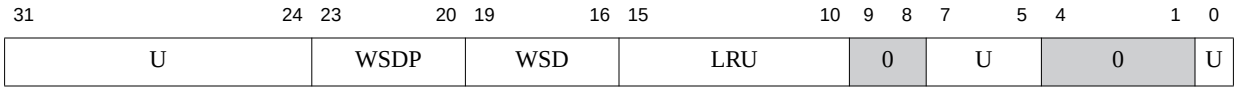
B.4.6 Cache registers in special diagnostic modes

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 3.4.10 “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 B-4 Fields in the TagLo Register (*ErrCtl[WST]* set)



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 C

MIPS® Architecture quick-reference sheet(s)

C.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 C.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 C.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		

C.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.

C.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 C.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 core 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**.

C.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 four 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[30]; 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.

30. 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 you read a value of "2", then *Count* increments every 2 cycles, at half the pipeline clock rate).

C.3 FPU changes in Release 2 of the MIPS32® Architecture

The main change is that a 32-bit CPU (like the 24K core) 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 C.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 24K core is described in Chapter 6, "Floating point unit" on page 61.

Appendix D

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
2.00	15th March 2004	First generally available version, to coincide with general availability of the 24K core.
2.01	22nd March 2004	*Config7[ES]* now defaults to zero.
2.02	4th June 2004	Improvements to the multiply/divide unit timings table and description.
2.03	29th September 2004	For RTL MR1 release. Made the debug section into a chapter in its own right. Added sections on scratchpad and EJTAG. Other minor fixes.
3.00	22nd April 2005	Added I-side scratchpad information to Section 3.6, "Scratchpad memory/ SPRAM" and Section 7.2 “PDtrace™ instruction trace facility” section, for the MR3 release of the 24K core. Added description of the *Config1-3* CP0 registers.
3.01		Minor typographical fixes.
4.00	1st July 2005	Update for maintenance release of the MIPS 24Kc core family. Minor updates on EJTAG debug and PDtrace sections.
4.10	21st December 2005	Update for maintenance release of the MIPS 24Kc core family. Added 8K cache option and improved description of scratchpad RAM.
4.20	22nd June 2006	Update for maintenance release.
4.51	23rd April 2007	L2 cache option documented. Updated for core version 3.7 Change bars are against 4.20
4.53	10th September 2007	Minor fix (FSB/LSB configurability missed in intro).
4.61	20th September 2007	Candidate for v4.0 release of the 24K core. Changes include: • New CP0 register, see Section B.4.2 “The UserLocal register”. • Alias-proof I-cache operations, see Section 3.4.8 “Cache aliases”. • Can **`wait`** with interrupts disabled, see Section 5.5 “Saving Power”. • The L2 access registers are renamed to *L23TagLo* etc (used to be “STagLo” etc). • Miscellaneous fixes. Change bars are vs 4.51.

Revision	Date	Description
4.62	31st October 2007	Minor cleanup for release • Add notes on L2 feature enhancement - 64B lines • Added missing UserLocal references Change bars are vs. 4.51
4.63	19th December 2008	• Exception table wrongly indicated machine check could not happen • Fixed read value of *CCRes* • Added example idle loop code making use of *Config7[WII]*