Programming the MIPS32® 24KE™ Core Family

Document Number: MD00458
Revision 1.10
December 21, 2005

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

Template: B1.15, Built with tags: 2B MIPS32 PROC

Table of Contents

Figures

Tables

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").

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 24KE™ core. So this document concentrates on where a MIPS 24KE family core behaves differently from its predecessors. That's either:

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

  But the summary is too brief to program from, and the details are widely spread; so you'll find a shortform presentation of the changes here in Chapter 9 "What's new in 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 [24KUSER] 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 24KE core.

All 24KE 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 24KE core CP0 registers in Appendix B "CP0 registers by name and number".

Chapters of this manual

- Chapter 2 "Initialization and identity": what happens from power-up? boot ROM material, but a good place to cover how you recognize hardware options and configure software-controlled ones.
- Chapter 3 "Memory mapping, memory access, caches and translation": everything about memory accesses.
- Chapter 4 "Kernel programming of the 24KE™ core": 24KE-core-specific information about privileged mode programming.
- Chapter 6 "Programming the 24KE™ core in user mode": features relevant to user-level programming; multiply timing, hardware registers, prefetching.
- Chapter 7 "The MIPS32® DSP ASE": A brief summary of the MIPS DSP ASE, available on members of the 24KE core family.
- Chapter 8 "Floating point unit": the 24KE core's floating point unit, available on models called 24KEf™.
- Chapter 9 "What's new in Release 2 of the MIPS32® Architecture?": a programmer's guide to new features.
- Appendix A "References": more reading to broaden your knowledge.

[1] References (in square brackets) are listed in Appendix A "References".

- Appendix B "CP0 registers by name and number": all the registers, and references back into the main text.
- Appendix C "User instructions added for the MIPS32® Architecture": instructions which aren't in the MIPS I ISA.

Conventions

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

Instruction mnemonics are in **`bold monospace`**; register names in `small monospace`. Register fields are shown after the register name in brackets, so the interrupt enable bit in the status register appears as `Status[IE]`.

1.1 24KE™ core features

All 24KE 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 D-cache configuration can be made free of "cache aliases" - see Section 3.3.2 "Cache aliases". The option is selected when the "cache wrapper" is defined for the 24KE core in your design - ask one of your chip designers.

  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 24KEPro 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, 24KE 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.
- *DSP ASE*: available only in 24KE family cores, this is a lot of new computational instructions with a fixed-point math unit crafted to speed up popular signal-processing algorithms, which form a large part of the computational load for voice and imaging applications. Some of these functions do two math operations at once on two 16-bit values held in one 32-bit register.

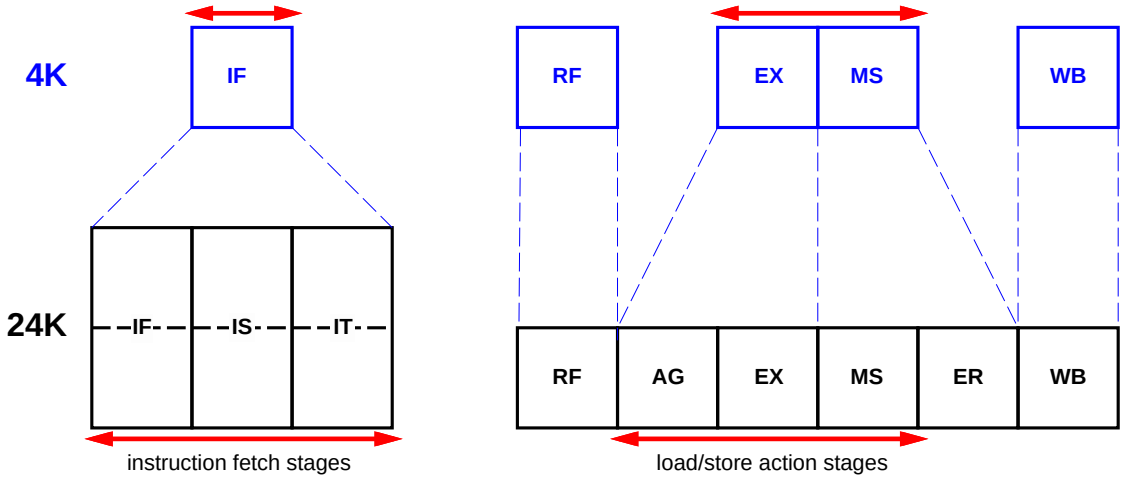
  There's a guide to the DSP ASE in Chapter 7 "The MIPS32® DSP ASE" and the full manual is [MIPSDSP].

1.2 A brief guide to the 24KE™ core implementation

All 24KE 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 24KE core is mainly different from its predecessors in the mechanisms used to mitigate those effects.

Figure 1-1 Pipeline differences between the 24KE™ and 4K™ core families



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 24KE 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 24KE core's mechanisms to ameliorate branch and load penalties.

1.2.1 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 24KE 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 [24KUSER], 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.

  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".

---

[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 24KE core - do it that way.

Note that the "likely" part of the name has nothing to do with branch prediction; the 24KE core's branch prediction system treats the "likelies" just like any other branches.

[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.

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

1.2.2 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 24KE 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 24KE 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.

1.2.3 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)* : you're unlikely to reach this limit unless you are using prefetch or otherwise deliberately optimizing loops. If a series of prefetches use all available resources, the next unrelated load-miss will stall the pipeline. It's likely to be good practice for code making conscious use of prefetches to ration itself to three outstanding operations.
- *Non-blocking loads to registers (four)* : there are just four data structures which remember outstanding loads 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+)* : the 24KE 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.

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 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.

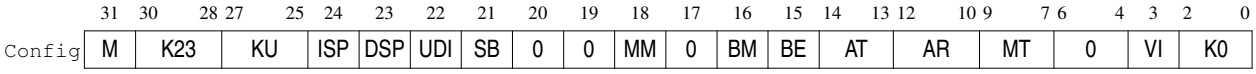
The 24KE core also uses `Config7` for some implementation-specific settings (which most programmers will never use).

Broadly speaking the registers have these roles:

`Config`	A mix of historical and CPU-dependent information, described in Figure 2-1 below. Some fields are writable.
`Config1`	Read-only, strictly to the MIPS32 architecture. Primary cache configuration and basic CPU capabilities,
`Config2`	Read-only, strictly to the MIPS32 architecture. Secondary and tertiary cache configuration, if fitted (they're not in the 24KE core).
`Config3`	Read-only, strictly to Release 2 of the [MIPS32] architecture. More CPU capability information.
`Config7`	24KE-core-specific, some fields are writable. Software disables for CPU pipeline performance features. Useful to help measure the contribution of these features to system performance. In addition, these could be used as software workarounds if very early parts have bugs. See Figure 2-4 below.

The first 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`: If your 24KE core-based system uses fixed mapping instead of having a TLB, you set the cacheability attributes of chunks of the memory map by writing these fields. If you have a TLB, these fields are unused (write only zeroes to them).

`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)

Down at the bottom of the register `Config[K0]` sets the cacheability of kseg0, but it would be very unusual to make that anything other than cacheable.

`ISP, DSP`: read 1 if I-side and/or D-side scratchpad (SPRAM) is fitted, see Section 3.4 "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, as is done by 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).

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 doesn't affect 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 -

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

AR: Architecture revision level -

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

MT: MMU type (all MIPS Technologies cores are 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: is the fixed kseg0 region cached or uncached? And if cached, how exactly does it behave - this field is encoded just like the "cache coherency attribute" field of a TLB entry, as it shows up in the `EntryLo0-1` register.

Caching attributes

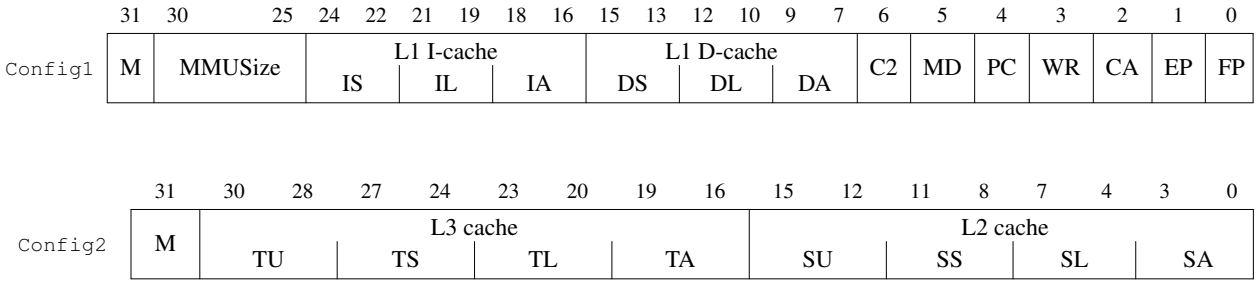
The fixed fields `Config[K0]`, `Config[KU]` and `Config[K23]` share a three-bit encoding with the caching attribute field in the TLB entries, described in Table 3-2 in Section 3.3.1 "Cacheability options".

Post-reset value of Config[K0] (starts uncached)

The power-on value of this standard field is not mandated by the [MIPS32] architecture; but the 24KE 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.

Config1 and Config2 - TLB and MMU information

Figure 2-2 Fields in the Config1-2 registers



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

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

`L1 I-cache, L1 D-cache`: 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]`: 1 if there's a coprocessor 2 fitted (that would be a customer-designed coprocessor).

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

`Config1[PC]`: there is at least one performance counter implemented, see Section 5.4 "Performance counters".

`Config1[WR]`: reads 1 because the 24KE core always has watchpoint registers, see Figure 5-12 "Fields in the WatchLo/WatchHi registers".

`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 5.1 "EJTAG on-chip debug unit".

`Config1[FP]`: a floating point unit is attached.

`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.

Config3

Figure 2-3 Fields in the Config3 register

	31	30 ... 11	10	9 ... 7	6	5	4	3	2	1	0
Config3	M	0	DSPP	0	VEIC	VInt	SP	0	MT	SM	TL

Fields shown in Figure 2-3 include:

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

DSPP: reads 1 if the MIPS DSP extension is implemented - as it is for the 24KE core, as described in Chapter 7 "The MIPS32® 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 9.5 "Release 2 - enhanced interrupt system(s)".

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

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

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

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

TL: reads 1 if your core is configured to do instruction trace (see Section 5.2 "PDtrace™ instruction trace facility".)

Config7

The Config7 register is used as the place for core-specific configuration information. It's also used for some controls for performance analysis and (possibly) workarounds. It's likely to change late and often, so if you have any trouble with these definitions consult [24KUSER]:

Figure 2-4 Fields in the Config7 register

	31 ... 18	17	16	15 ... 9	8	7	5	4	3	2	1	0
Config7	0	FPR	AR	0	ES	0		ULB	BP	RPS	BHT	SL

The active fields in config7 are:

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 (see Section 3.3.2 "Cache aliases").

All the remaining fields are read/write, and control various functions:

Config7[ES] defaults to zero. If set, the **sync** instruction will be signalled on the core's OCP interface as an "ordering barrier" transaction. The transaction is an extension to the OCP standards, and system controllers which don't support it will typically under-decode it as a read from the boot ROM area. But that's going to be quite slow: so set this bit only if your system understands the synchronizing 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[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 jump stall as above.

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).

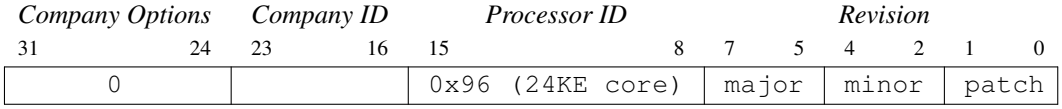
Config7[BHT]
when set, the branch history table is disabled and all branches are predicted taken. This bit is don't care if `config7[BP]` is set.

Config7[SL] when set, disables non-blocking loads. Normally the 24KE 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.

2.2 PRId register value

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 its 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 a directed software probe.

Figure 2-5 24KE processor ID (PRId) register



The revision field is divided into major and minor; a nonzero ‘‘patch’’ revision number is for use when something changes in an old revision of the core.

The ‘‘company options’’ and ‘‘company ID’’ field may be set to any value by whoever configures the 24KE core into their system on a chip. The value ‘‘1’’ for company ID is reserved to MIPS Technologies.

Memory mapping, memory access, caches and translation

Sections include:

- Section 3.1 “The memory map”: memory regions and what they do.
- Section 3.2 “Reads, writes and synchronization”: delays, sequencing and special access types.
- Section 3.3 “Caches”: shape and size, cache aliases, write protocols.
- Section 3.5 “The TLB and translation”: memory management issues.

3.1 The memory map

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

The TLB version uses the memory map described by the [MIPS32] architecture, which will be familiar to anyone who has used a 32-bit MIPS CPU. It support 32-bits of addressable locations on the system interface. More information about the TLB in Section 3.5 “The TLB and translation”.

Fixed mapping option

To save chip area for applications not needing a full TLB, a 24KE core-based system can be built with a simple fixed mapping (“FMT”) memory translator, which plays the same role. Virtual address ranges are hard-wired to particular physical addresses, and cacheability options are set through CP0 registers, as summarized in Table 3-1:

Table 3-1 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]

Note that even in fixed-mapping mode, the cache error status bit Status[ERL] still has the effect (required by the MIPS32 architecture) of usurping 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.

3.2 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.

3.2.1 Non-blocking loads, and sequencing of loads and stores

A 24KE CPU has non-blocking loads; that is, the pipeline does not stall while load data arrives. Instead, the target register of the load is marked and remembered; execution continues until an operation wants to consume the value returned by the load. This has a number of possibly surprising consequences:

- *Number of non-blocking loads which may be pending* : the CPU supports four pending loads - either uncached loads or resulting from cache misses. Only a fifth load will cause it to stall.

  Of course, if any instruction references a register which is the subject of a pending load, everything stops until the data arrives.

- *Hit-under-miss* : the D-cache continues to supply data on a hit, even though one or more cache refills may be pending.

  So it is quite normal for a read which hits in the cache to be completed before a previous read which missed.

- *Write-under-miss* : the CPU pipeline continues and can generate stores even though a read is pending. The 24KE 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 24KE core can continue to run until it accumulates up to four pending read operations.
- *Core interface ordering* : even 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 24KE core - as is permitted by [MIPS32] - makes no promises about the order in which reads and writes happen at the system interface.

  If you want to be sure that some other agent in the system sees a pair of transactions in the order of the instructions that caused them, you should put a **sync**[4] instruction between the instructions.

3.2.2 Uncached accelerated writes

The 24KE 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.

Such regions are uncached for read, and partial-word writes have ‘‘unpredictable’’ effects - don’t do them. But sequential word stores in such regions are gathered into cache-line-sized chunks each of which is written with a single burst cycle on the CPU interface. This burst is marked by per-word ‘‘valid’’ bits, so that unwritten memory words are not corrupted.

The burst write is normally performed when software writes to the last location in the memory block 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.

3.2.3 Bus parity error/Cache Error

The 24KE core does not check parity on data (or control fields) from the external interface.

CacheErr register

Some implementations - it’s a build-time option - will include parity bits in the cache. At a system level, a cache parity exception is deeply fatal. Diagnostics writers will probably find the `CacheErr` register’s extra information useful, but it’s not further described here.

ErrCtl register

This register controls parity protection of the L1 caches (if it was configured in your core in the first place) and provides for software testing of the whole cache array, including the otherwise-inaccessible way-selection RAM.

Figure 3-1 Fields in the ErrCtl register

	31	30	29	28	27	26 … 19	18 … 13	12 … 4	3 … 0
ErrCtl	PE	PO	WST	SPR	PCO	0	PCI	PI	PD

In summary: running software should set this register to `0x8000.0000` to enable cache parity checking, and to zero otherwise.

Other uses are more obscure, but 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. Mainly for diagnostic/test purposes.

After setting this bit you can use **cache Index_Store_Tag/cache Index_Store_Data** 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]`.

[4] Note that **sync** is described as only working on ‘‘uncached pages or cacheable pages marked as coherent’’. It’s fair to assume that **sync** also acts as a synchronization barrier to the effects produced by routine cache-manipulation instructions - hit-writeback and hit-invalidate.

WST: test mode for the way-selection RAM[5].

SPR: when set, index-type **cache** instructions work on the scratchpad/SPRAM, if fitted - see Section 3.4 "Scratchpad memory/SPRAM".

PCO/PCI: precode override and data. Used for diagnostic/test of the I-cache feature which partially decodes instructions at cache refill time. Not properly documented here.

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

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 is not 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.

The bus error is imprecise; that is, EPC does not necessarily point to the instruction causing the load (except for one special case: a bus error is precise when it occurs on an uncached or cache-missing instruction fetch). Load errors are particularly imprecise, since the (non-blocking) load which caused it may have happened a long time ago.

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

3.3 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 24KE 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 3.3.2 "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 24KE core's primary caches are 4-way set associative.

But don't hardwire any of this information into your software. Instead, probe the Config1 register defined by [MIPS32] to determine the shape and size of the cache.

3.3.1 Cacheability options

Any read or write made by the 24KE 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 3-2 shows the code values used in EntryLo[C] and in the Config entries used to set the behavior of regions with fixed mappings (the latter are described in Figure 2-1 "Fields in the Config register".)

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

Table 3-2 Cache attributes for TLB and fixed-segment fields

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 3.2.2 "Uncached accelerated writes".

[5] A real explanation of this is beyond the scope of this manual...

3.3.2 Cache aliases

24KE core has caches which are virtually indexed. 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's not a problem for the normal operation of caches for read-only data, but 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 they're distinguished through the physical tag.

The 24KE 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 (or 8Kbyte cache) is alias-free. 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. You 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[6]...

So the 24KE core contains logic to make the popular 32Kbyte D-cache alias-free (effectively one index bit is from the physical address, and some ingenious tricks used to prevent that slowing the whole process excessively). Consult your local SoC designer about how the core has been configured.

A 32Kbyte I-cache, or any 64Kbyte I- or D-cache, are subject to aliases.

3.3.3 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[7] - but note that **`synci`** won't do the job, and should not be used on data/instruction locations which are cache-locked.

3.3.4 The cache instruction and software cache management

The 24KE core implements all the **`cache`** instructions defined by [MIPS32] (whether said there to be ''required'', ''recommended'' or ''optional''.)

Before any **`cache`** instructions is allowed to execute, all initiated refills are completed and stores are sent to the write buffer.

[6] There's a fair amount of rather ugly code in the MIPS Linux kernel to work around aliases, and there are some corner cases about mapping files which remain unfixed.

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

The **synci** instruction (new to the MIPS32 Release 2 update) provides a clean mechanism 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: a D-cache writeback followed by an I-cache invalidate.

3.3.5 Cache initialization and tag/data registers

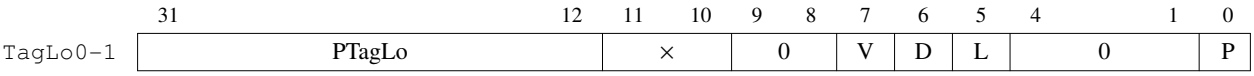
The TagLo0-2 registers are used for staging tag information being read from or written to the cache (the 24KE core has no "TagHi" registers). [MIPS32] declares that the contents of these registers is implementation dependent, so they need some words here.

TagLo0 is used for the I-cache and TagLo1 for the D-cache. TagLo2 is reserved for secondary cache management, and is not described further.

But the architecture does require that you can write all-zeros to both registers 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.

You'll need to consult the full [24KUSER] manual to write cache diagnostics, and you probably don't even need the field names for normal system programming. But Figure 3-2 is a basic picture of the registers, showing both descriptive names and the field names as used in the full manual.

Figure 3-2 Fields in the TagLo0-1 Registers



TagLo0-1 can be used in two special modes:

- *Way Select RAM access* : when ErrCtl[WST] is 1, the appropriate TagLo register's fields change completely, as shown in Figure 3-3 below;
- *Scratchpad RAM control access* : when ErrCtl[SPR] is set, I-side/D-side cacheops (respectively) are used to access a control field for "scratchpad RAM", if it's fitted. See Section 3.4 "Scratchpad memory/SPRAM" below.

But let's look at the standard fields first:

PTagLo: the cache address tag - a physical address because the 24KE core's caches are physically tagged. It holds bits 31–12 of the physical address - the low 12 bits of the address are implied by the position of the data in the cache.

×: a field not described for the 24KE core but which might not always read zero.

V: 1 when this cache line is valid.

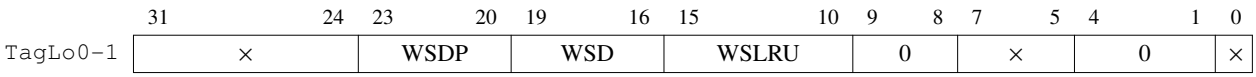
D: 1 when this cache line is dirty (that is, it has been written by the CPU since being read from memory).

L: 1 when this cache line is locked, see Section 3.3.3 "Cache locking".

P: parity bit for tag fields other than the TagLo[D] bit, which is actually held separately in the "way-select" RAM. When you use the TagLo register to write a cache tag with **cache IndexStoreTag** the

TagLo[P]: bit is generally not used - instead the hardware puts together your other fields and ensures it writes correct parity. However, it is possible to force parity to exactly this value by first setting ErrCtl[PO].

Figure 3-3 Fields in TagLo0-1 when used for way-select RAM



When `ErrCtl[WST]` is set, **`cache IndexLoadTag`** and **`cache IndexStoreTag`** operations read/write the separate ''way-select RAM'' used in the 24KE core's caches. Then the fields in `TagLo0-1` change to those shown in Figure 3-3 above. These are:

`WSDP`: parity check for each of the ''dirty'' bits. It's like the regular tag parity bit, in that this field is not normally used when you're writing into the way-select RAM. If you want to force these values in, you need to set `ErrCtl[PO]` to 1.

`WSD`: dirty bits - found on D-side only, so not in `TagLo0`.

`WSLRU`: LRU bits.

3.4 Scratchpad memory/SPRAM

Most of MIPS Technologies' cores can be equipped with a modestly sized high speed on-chip data memory, fitted to a special quasi-cache interface, called ''scratchpad RAM'' or ''SPRAM''. SPRAM is connected alongside the I- 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, but provides a ''reference design'' for both ISPRAM and DSPRAM. If you follow 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:

- *Dedicated high-speed memory* : small SPRAM arrays run with cache timing. Larger arrays may require one or more clocks of extra latency, but are still very fast compared with any memory access through an OCP interface[8].

  SPRAM can be made much 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 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 the OCP interface. OCP masters will see it just as a chunk of memory which can be read or written.

  On the I-side, an externally-writable ISPRAM provides a way in which you can load firmware into a 24KE family core so it has no need of external bootstrap memory (for some tiny, dedicated applications you might not need any external instruction memory at all).

  On the D-side, because SPRAM stands in for the cache, data passed through the DSPRAM 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 work too with MIPS Technologies' reference design, but may not do so on other implementations - better to keep it cached). 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.

[8] In the 24KE family, the way the instruction fetch unit works means that any ISPRAM which can't deliver data at cache speed causes a pipeline ''replay'' within the fetch unit, which is relatively expensive, reducing the advantage gained. Data-side SPRAM may be made multi-cycle without this sort of problem.

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], and (in the case of ISPRAM) to load instructions into it.

Probing and setting up SPRAM

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 "Fields in the Config register".

The MIPS Technologies reference design requires that you can query the size and adjust the location of scratchpad RAM through "cache tags". To access these SPRAM "tags", first set the `ErrCtl[SPR]` bit (see Figure 3-1 "Fields in the ErrCtl register" above).

Now a D-side **cache Index_Load_Tag_D, 0** instruction[10] fetches half the configuration information into `TagLo1`, and a **cache Index_Load_Tag_D, 8**† gets the other half into `TagLo1`. The corresponding operations directed at the primary I-cache read the halves of the I-side scratchpad tag, this time into `TagLo0`. The fake tags for I-side and D-side SPRAM have the same format; the information appears in `TagLo0/1`'s fields as shown in Figure 3-4.

Figure 3-4 SPRAM (scratchpad RAM) configuration information in TagLo0/1

	31 … 12	11 … 10	9 … 8	7	6	5	4 … 1	0
TagLo1	PTagLo	×	0	V	D	L	0	P

SPtag #0	base address[31:12]	0	En	0
SPtag #1	size of region in 4KB units	0		

Where:

- *base address[31:12]*: (writable) the high-order bits of your chosen physical base address of this chunk of SPRAM;
- *En*: from power-up this bit is zero, and so long as it stays that way the SPRAM acts as though it isn't there;
- *size of region in 4KB units*: (you should only read this) the number of page-size chunks of data mapped. In fact this number will be a power of 2 between 1 and 256, corresponding to a memory size between 4KB-1MB.

Some MIPS cores using this sort of tag setup have permitted multiple scratchpad regions indicated by two or more of these tag pairs. But the reference design for the 24KE core can only at most one I- and one D- region.

You can load software into the ISPRAM using cacheops. Each pair of instructions to be loaded are put in the registers `DataHi0`/`DataLo0`, and then you perform an index store data cache instruction **cache Index_Store_Data_I** at the appropriate index. The two data registers work together to do a 64-bit transfer (the 24KE core's instruction memory really is 64 bits wide), so for a CPU configured big-endian the first instruction in sequence is loaded into `DataHi0`, but for a CPU configured little-endian the first instruction is loaded into `DataLo0`.

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

[9] All these details apply to MIPS Technologies' reference SPRAM designs; other implementors are urged, but not obliged, to follow these conventions.

[10] The instructions here are written using C preprocessor definitions from [m32c0.h].

† The "8" steps to the next feasible tag location. Why 8? It's to do with the 24KE core's 64-bit wide (ie 8 byte wide) interface to cache data.

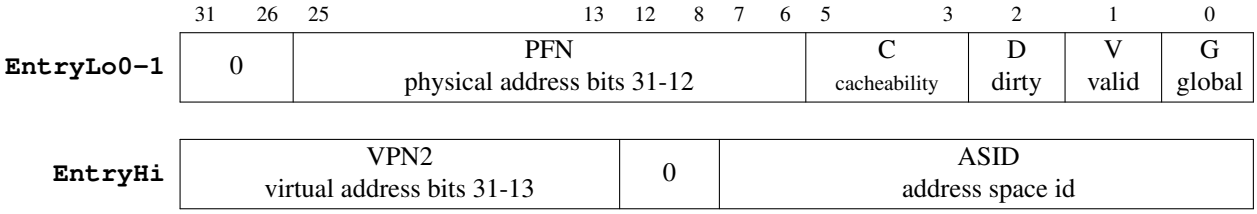
3.5 The TLB and translation

The TLB is the key piece of hardware which MIPS architecture CPUs have for memory management. It's a table whose input is virtual addresses together with the "address space identifier" from `EntryHi[ASID]` and whose output is a physical address plus cacheability attributes. System software maintains the TLB as a cache of a much larger number of possible translations, and a special-cased handler for the exception raised when there's no suitable translation in the TLB provides adequate performance. That's far too complicated to cover here; for a better description see [SEEMIPSRUN], and for full details see [MIPS32].

When you're really tuning out the last cycle, you need to know that in the 24KE 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 (or generically, they're "micro-TLBs" or "uTLB"). 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 when required, and automatically cleared whenever the TLB is updated[11]. It costs just three extra clocks to refill the uTLB for any access whose translation is not already in the appropriate uTLB.

The TLB is accessed through staging registers which between them represent all the fields in each TLB entry; they're called `EntryHi` and `EntryLo0-1`, and the fields are shown in Figure 3-5.

Figure 3-5 Fields in the TLB (EntryLo0-1 and EntryHi) registers



TLB initialization and duplicate entries

Since the TLB is a fully-associative array and entries are written by index, it's possible to load a duplicate entry. Duplicate entries will certainly cause chaos and - in some designs - might even cause hardware damage.

MIPS Technologies cores peek in the TLB before writing a new entry; they take a "machine check" exception if the write would have created a duplicate entry.

Earlier MIPS Technologies cores required particular care. When an OS is initializing a TLB for the first time (usually by filling it with dummy entries) 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: but that's not essential on the 24KE core, where the machine check exception doesn't happen unless both the old and new entry are both marked as valid.

3.5.1 Page sizes

The 24KE core supports page sizes in power-of-four sizes from 4Kbytes to 256Mbytes; the page size (ie how many bits of the address are checked to see if there's a match) is an attribute stored independently by each TLB entry.

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 lead to a noticeable performance loss.

[11] ITLB entries don't contain the address-space ID ("ASID", held in `EntryHi[ASID]`.) Those entries would be misleading after you changed the ASID, so the ITLB is also automatically cleared if you write to `EntryHi`.

3.5.2 Cacheability attributes

A three-bit field in each TLB entry (represented by `EntryLo0-1.C`) is used to determine whether and how to cache data on I-fetches and loads, and how to handle writes. See Section 3.3.1 ‘‘Cacheability options’’ in the section on caches above.

Kernel programming of the 24KE™ core

Sections include:

- Section 4.1 “Handling exceptions and interrupts”: the timer, new interrupt features and shadow register sets.
- Section 4.2 “Software-managed hazards”: and how they’re much cleaner because the 24KE core conforms to Release 2 of the MIPS32 architecture.
- Section 4.3 “Reducing power consumption”: features available.

4.1 Handling exceptions and interrupts

4.1.1 The counter/timer

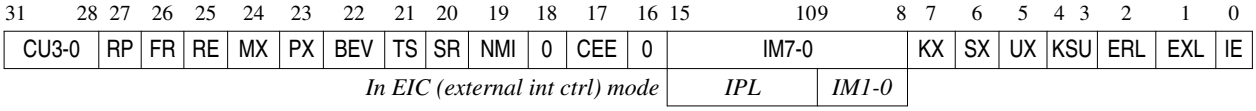
The in-CPU counter/timer is required by the [MIPS32] architecture. It may run at the pipeline clock, or at some divider; to find out how fast it runs use **`rdhwr CCRes`**, as described in Section 9.1.2 “Release 2 of the MIPS32® Architecture - Hardware registers from user mode” [12].

Some systems may well reduce power by slowing the core input clock (either using the documented `Status[RP]` bit or otherwise); in that case, the counter will slow down too; `Count` ultimately runs off the same clock as the rest of the CPU.

The counter does, however, keep running during the sleep period following a **`wait`** instruction.

4.1.2 Status register

Figure 4-1 All status register fields



The status register - all of whose fields are shown in Figure 4-1 - is the main home for CPU state flags (privilege level, interrupt enables, coprocessor enables). The 24KE family `Status` has no non-standard fields. There are, though, some fields where its behavior requires some additional clarification, and they’re shown in Table 4-1.

[12] In the first version of the 24KE core `Count` runs at half the pipeline rate; but it’s better to read the register, once you’ve established that the CPU conforms to Release 2 of the MIPS32 architecture.

Table 4-1 Non-standard fields in the Status register

Name	*Bits*	*Description*
RP	27	Reduced power - standard field. It's not connected inside the 24KE core, but the state of the RP bit is available on the external core interface as the *SI_RP* signal. The 24KE core uses clocks generated outside the core, and this could be used in your design to slow the input clock(s).
SR	20	MIPS32 architecture "soft reset" bit: the 24KE core's interface only supports a full external reset, so this always reads zero.
CEE	17	CorExtend Enable: read/write bit. Set zero to disable the CorExtend block, but only *if it needs it*. CorExtend blocks will typically 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. **Note**: this field is not really CPU-dependent; it will be defined as part of CorExtend in future.

4.1.3 Interrupts in Release 2 of the MIPS32® Architecture

Although now a standard part of the [MIPS32] architecture, the interrupt vectoring ("VI") and hardware priority ("EIC") features are quite new to the MIPS32 architecture, so are described in Section 9.5 "Release 2 - enhanced interrupt system(s)".

4.1.4 Shadow registers

Your 24KE core-based system can be configured with shadow register sets; [MIPS32] (Release 2) provides mechanisms to find out how many sets there are, and to manipulate them. See Section 9.6 "Shadow registers" in the section about Release 2 features.

4.2 Software-managed hazards

With the MIPS32 Release 2 update, you're at last freed from having to pad code with varying numbers of **`nop`** or **`ssnop`** instructions to ensure that the state change made by some privileged instruction (probably one writing a CP0 register) has taken proper effect.

From Release 2 of the MIPS32 specification [MIPS32] this is no longer CPU-dependent; you can read about it in Section 9.3 "Hazard barriers - no more CPU-dependent no-ops".

4.3 Reducing power consumption

The 24KE core is designed to be frugal on power in comparison to its high performance.

There are just two software controls on power consumption in the core (though your system may have more controls in external logic, perhaps accessible through some outside-the-core I/O register):

- 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.

- The reduced-power `Status[RP]` bit in the status register (see Table 4-1 "Non-standard fields in the Status register"). This bit has no direct effect on the core, but is exported as a signal which may be respected by the clock generator - so if it does have an effect, it will affect absolutely everything in the 24KE core. Ask your system-on-a-chip designer.

Debug and visibility facilities

Increasingly, our cores are equipped with features to help during software development. The 24KE core comes with:

- Section 5.1 "EJTAG on-chip debug unit": which is described at length in [EJTAG]. The description here is not complete. It's intended to let you understand EJTAG and appreciate what options are available with the 24KE core. To use the debug unit to the full you need a suitable hardware "probe"; but you can also use EJTAG facilities for in-system software debug, and the description here would help you do that.
- Section 5.3 "Watchpoint registers": non-EJTAG CP0 registers, which can be programmed to generate an exception on an I-fetch, load or store from a particular address range. Most often used from inside a debugger.

  The CP0 watchpoints are independent of the similar facility inside the EJTAG unit.
- Section 5.4 "Performance counters": CP0 registers which count one of a large variety of interesting events, allowing you to get a statistical view of what low-level actions are contributing to the run-time of your program.

  Performance counters are emerging as a significant and useful tool for CPU architects, system designers and programmers, so the 24KE core has a rather richer set than previous cores from MIPS Technologies.

  Performance counter support generally needs to be built into the OS (to associate counts with the right tasks).
- Section 5.2 "PDtrace™ instruction trace facility": if your system is built with the "PDtrace™" option, this provides a mechanism for extracting instruction traces from the running CPU, so suitably-prepared external hardware can see exactly what code was executed.

  There's a lot of subtleties in the PDtrace specification which are really only important if you're building a debug probe, and are not important to local software. Those aren't described here: you'd perhaps get some help from documentation provided with your PDtrace probe and host software.

5.1 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[13].

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 5.1.4 "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 5.1.2 "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 "drseg";
- 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 Section 5.1.11 "PC Sampling with EJTAG" below.

[13] It can actually be quite useful to provide EJTAG with its own pins, if your package permits.

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 24KE 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.

5.1.1 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 and EJTAG unit agree a number of JTAG "instructions", each of which typically reads/writes one of a number of registers. EJTAG's instructions are shown in Table 5-1.

Table 5-1 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 5-6 below.
`EJTAG_ADDRESS` `EJTAG_DATA`	(read/write) together, allow the probe to respond to instruction fetches and data reads/writes in the magic "dmseg" region described in Section 5.1.4 "The "dseg" memory decode region".
`EJTAG_CONTROL`	Package of flags and control fields for the probe to read and write; see Figure 5-7 below.
`EJTAGBOOT` `NORMALBOOT`	The "EJTAGBOOT" instruction causes the next CPU reset to lead to CPU booting from probe; see description of the `EJTAG_CONTROL` bits `ProbEn`, `ProbTrap` and `EjtagBrk` in the notes on Figure 5-7. The "NORMALBOOT" instruction reverts to the normal CPU bootstrap.
`FASTDATA`	Special access used to accelerate multi-word data transfers with probe. The probe reads/writes the 33-bit register formed of `EJTAG_CONTROL[PrAcc]` with `EJTAG_DATA`.

`TCBCONTROLA`
`TCBCONTROLB`
`TCBADDRESS`
	Access registers used to control "PDtrace" instruction trace output, if available. See Section 5.2 "PDtrace™ instruction trace facility", but these JTAG-accessible registers are not documented in this manual.

5.1.2 Debug mode

A special CPU state; the CPU goes into debug mode when it takes any debug exception - which can be caused by an **`sddbp`** 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 and controllable as `Debug[DM]` (see Figure 5-2 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 Section 5.1.2 "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 9-3 "Exception entry points".

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.

5.1.3 Single-stepping

When the single-step bit Debug[SSt] is set and control returns from debug mode with a **deret**, the instruction selected by DERET will be executed in non-debug context[14]; then a debug exception will be taken on the very next instruction to be fetched in sequence. Since 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.

[14] If DERET points to a branch instruction, both the branch and branch-delay instruction will be executed normally.

5.1.4 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 Figure 5-1.

Figure 5-1 EJTAG debug memory region map (“dseg”)

Virtual Address	*Region/sub-regions*		*Location/register*	*Virtual Address*
0xE000.0000				0xE000.0000
0xFF1F.FFFF				0xFF1F.FFFF
0xFF20.0000	kseg3	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 Figure 5-1:

- *dseg* : is the whole debug-mode-only memory area.

  It’s possible for debug-mode software to read the “kseg3”-mapped locations “underneath” by setting `Debug[LSNM]` (see Figure 5-2 “Fields in the EJTAG CP0 Debug register” below).

† If there’s no probe-supplied debug exception handler - a condition notified by the probe itself using the `EJTAG_CONTROL[ProbTrap]` bit - the debugger entry point will be at `0xBFC0.0480` (near the other “ROM” exception entry points).

- *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.

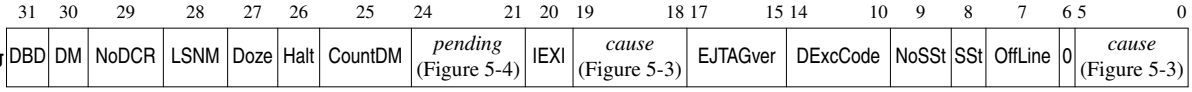
5.1.5 EJTAG CP0 registers, particularly Debug

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);
- DSAVE 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 5-3, information about regular exceptions which want to happen but can’t because you’re in debug mode in Figure 5-4, and everything else. The ‘‘everything else’’ category includes the most important fields and comes first, in Figure 5-2.

Figure 5-2 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 24KE core, so you get to choose. On 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 5-4 below.

`EJTAGver`: read-only - tells you which revision of the specification this implementation conforms to. On the 24KE 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]`, which is all described in [MIPS32V3].

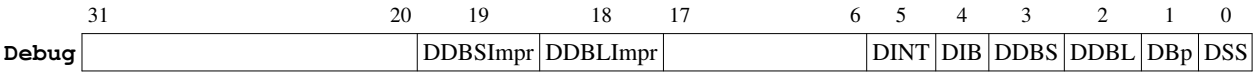
`NoSSt`: read-only - reads 0 because single-step is implemented (it always is on MIPS Technologies cores).

`SSt`: set 1 to enable single-step.

`OffLine`: often not implemented in single-threaded CPUs, in which case it just reads zero.

If implemented it stops the CPU from running instructions except in debug mode. This is intended to allow a debugger to shut down some processors in a multi-CPU system, while still having debug-mode life in them. The condition can only be cleared in debug mode, so if you once do a **`deret`** you must wait for an external, *DINT*-initiated return to debug mode before you can rescue the CPU.

Figure 5-3 Exception cause bits in the debug register



`DDBSImpr`: imprecise store breakpoint - see Section 5.1.10 "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 (from EJTAG pin).

`DIB`: instruction breakpoint.

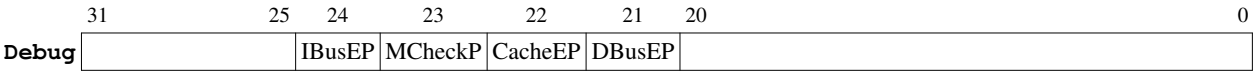
`DDBS`: precise store breakpoint.

`DDBL`: precise load breakpoint.

`DBp`: any sort of debug breakpoint.

`DSS`: single-step exception.

Figure 5-4 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 24KE 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 24KE core, so this is always zero.

`CacheEP`: cache error pending.

`DBusEP`: bus error on data access pending.

5.1.6 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 Figure 5-1 "EJTAG debug memory region map ("dseg")" (but only accessible if the CPU is in debug mode). The fields are in Figure 5-5:

Figure 5-5 Fields in the memory-mapped DCR (debug control) register

	31	30	29	28	18	17	16	15	10	9	8	6	5	4	3	2	1	0
DCR	0		ENM	0		DB	IB	0		PCS	PCR		0	INTE	NMIE	NMIP	SRE	PE

Where:

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

`DB/IB`: (read only) 1 if data/instruction hardware breakpoints are available, respectively. The 24KE core has either 0 or 2 data breakpoints, and either 0 or 4 instruction breakpoints.

`PCS, PCR`: `PCS` reads 1 if the PC sampling feature is available, as it can be on the 24KE core. Then `PCR` is a three-bit field defining the sampling frequency as one sample every $2^{(5+PCSR)}$ cycles. See Section 5.1.11 "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 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 24KE 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 5-7.

5.1.7 JTAG-accessible registers

We're wandering away from what is relevant to software here: this register is available for read and write only by the probe, and is 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 5-6); then there's the main CPU interaction control/status register `EJTAG_CONTROL` (Figure 5-7).

Figure 5-6 Fields in the JTAG-accessible ImpCode register

	31 29	28 25	24	23 21	20 17	16	15	14	13 1	0
JTAG ImpCode	EJTAGver 2 = 2.6	0	DINTsup *see note*	ASIDsize *see note*	0	MIPS16 1	0	NoDMA 1	0	MIPS32/64 0

Notes on the `ImpCode` fields:

`EJTAGver`: same value (and meaning) as the `Debug[EJTAGver]` field, see the notes on Figure 5-2.

`DINTsup`: whether JTAG-connected probe has a *DINT* signal to interrupt the CPU. Configured by your SoC designer (who should know) by hardwiring 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.

`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 (ie a fixed-mapping MMU).

`MIPS16`: 1 because the 24KE 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 24KE core's OCP interface).

`MIPS32/64`: the zero indicates this is a 32-bit CPU.

Figure 5-7 Fields in the JTAG-accessible EJTAG_CONTROL register

	31	30	29 28	23	22	21	20	19	18	17	16	15	14	13	12	11	4	3	2	0
EJTAG_CONTROL	Rocc	Psz	0		Doze	Halt	PerRst	PRnW	PrAcc	0	PrRst	ProbEn	ProbTrap	0	EjtagBrk	0		DM	0	

Notes on the fields:

`Rocc`: "reset occurred" - reads 1 while a reset signal is applied to 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.

`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*
×	0	Byte
00 10	1	Halfword
00	2	Word
00 01	3	Tri-byte (lowest address 3 bytes) 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, the probe should write the field and then poll for the change to be reflected in this register, which may take a short while.

`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 is conventionally 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.

`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` (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 reset-time value of all these three bits is preset from an internal bit set 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].

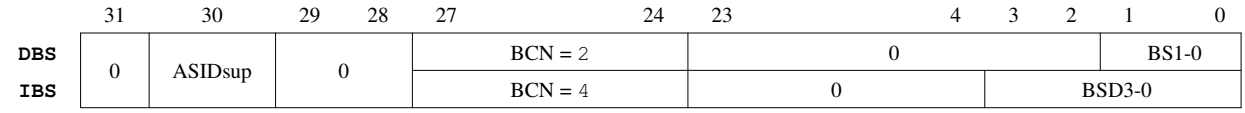
5.1.8 EJTAG breakpoint registers

It's optional whether the 24KE 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 24KE core - see Section 5.1.10 "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 Figure 5-1 "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 5-8.

Figure 5-8 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 24KE 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 5-9 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 instruction breakpoint may happen 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 5-9 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 31-64 to be matched for 64-bit load/stores: the 32-bit[15] 24KE has 64-bit load/store instructions for floating point.

[15] A JTAG hardware breakpoint for a real 64-bit CPU would have 64-bit DBVn registers, so

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 5-9.

Figure 5-9 Fields in the hardware breakpoint control registers (IBCn, DBCn)

	31 … 24	23	22 … 18	17 … 14	13	12	11 … 8	7 … 4	3	2	1	0
DBCn	0	ASIDuse	0	BAI 7-0	NoSB	NoLB	0	BLM7-0	0	TE	0	BE

	31 … 24	23	22 … 3	2	1	0
IBCn	0	ASIDuse	0	TE	0	BE

The fields are:

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

`BAI 7-0`: "byte (lane) access ignore"[16] - 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*[17] breakpoint on store/load respectively.

`BLM 7-0`: 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 5.2 "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!

wouldn't need `DBVHin`.

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

[17] "1-to-enable" would feel more logical. The advantage of using 0-to-enable here is that the zero value means you will break on either read or write, which is a better default than "never break at all".

5.1.9 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.
  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 the `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, ie `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.

5.1.10 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 24KE 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. The debugging software must cope.

5.1.11 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 5-5 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[18]; 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 5-5.

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.

[18] Since it runs continuously, it’s a good thing that from reset the sampling period defaults to its maximum.

5.2 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.

24KE 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 24KE 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 24KE core can be configured with up to 8 "trace triggers", allowing you to start and stop tracing based on EJTAG breakpoint matches: see Section 5.1.8 "EJTAG breakpoint registers" above and Figure 5-11 "Fields in the TraceIBPC/TraceDBPC registers" 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 5-10 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 to have a suitable debug/trace probe plugged into a connector on your board giving access to the JTAG/trace signals coming out of the chip containing the 24KE core. The probe will connect to some debug/trace host computer somehow...

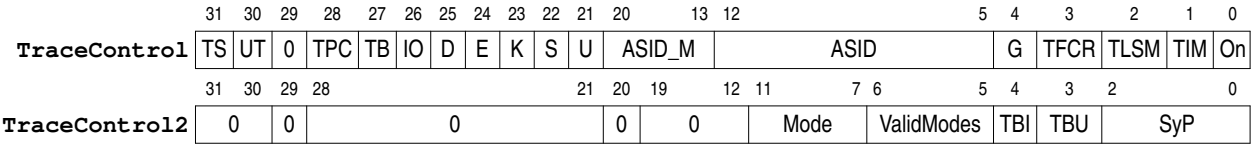
However, there are software-accessible PDtrace facilities, which can be used (for example) to get information from a suitably instrumented program or OS, via the probe, to your debug/trace software on your development system. This manual describes only the lowest-level building blocks as visible to software. For real hardware information refer to [PDTRACETCB]; for guidance about how to use the PDtrace facilities for software development see [PDTRACEUSAGE].

CP0 registers for the PDtrace™ logic

There are four:

- `TraceControl` and `TraceControl2`: allow the software to take charge of what is being traced.
- `UserTraceData`: allows software to send a "user format" trace record, which can be interpreted by suitable trace analysis software to build interesting facilities.
- `TraceBPC`: controls whether and how individual EJTAG breakpoint trace triggers take effect.

Figure 5-10 Fields in the TraceControl and TraceControl2 registers



`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.

`TPC`: turns on PC Sampling, where the current PC value is periodically sent to the trace memory.

`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[19]. 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

[19] Prior to v4 of the PDtrace specification, this field was in `TraceControl`, and was too small to allow all conditions to be specified independently.

TraceControl2[TBI,TBU]: best considered together, these read-only bits tell you whether there is an on-chip trace memory, probe 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 $2^{(SyP + 5)}$

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 5-11:

Figure 5-11 Fields in the TraceIBPC/TraceDBPC registers

	31	30 29	28	27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
TraceIBPC	0	0	IE	ATE						IBPC3	IBPC2	IBPC1	IBPC0
TraceDBPC			DE									DBPC1	DBPC0

In either TraceIBPC or TraceDBPC:

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

ATE: Read-only bit which lets you know whether the additional trigger controls such as ARM, DISARM, and data-qualified tracing (introduced in v4.00 of the PDtrace specification) are available - which they may be on the 24KE 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 24KE 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.
3	"Arm" the trace condition (allows a two-step process).
4	Stop trace (but only if armed.)
5	Start trace (but only if armed.)
6	Trace instructions which cause this trigger (but only if armed).
7	"disarm" the trace.

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 Section 5.1.6 "The DCR (debug control) memory-mapped register").

UserTraceData register

Write any 32-bit value you like here and the trace unit will send a "user" record (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!

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.

  But if the probe (via the JTAG interface) is in charge (that is, if `TraceControl[TS]` == `0`) then:
  - 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;
- 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).

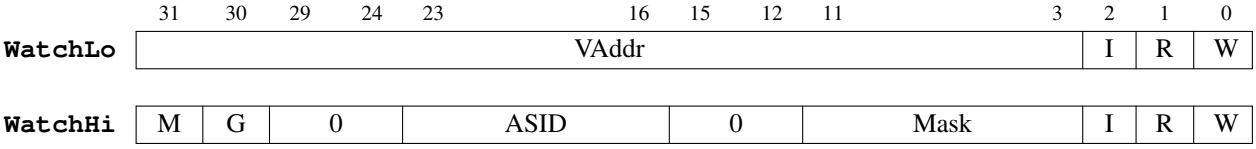
If there is more than one breakpoint match in the same cycle, an ‘‘on’’ trigger wins out over any number of ‘‘off’’.

5.3 Watchpoint registers

Each watchpoint is controlled by a pair of CP0 registers. The 24KE core has two instruction watchpoints `WatchLo0`/`WatchHi0` and `WatchLo1`/`WatchHi1`; and two data watchpoints `WatchLo2`/`WatchHi2` and `WatchLo3`/`WatchHi3`.

The CP0 watchpoints match on virtual addresses, not physical addresses.

Figure 5-12 Fields in the WatchLo/WatchHi registers



About Watchpoint register fields in Figure 5-12

`VAddr`: the address to match on, with a resolution of a doubleword.

`WatchLo[I], WatchLo[R], WatchLo[W]`: accesses to match: I-fetches, Reads (loads), Writes (stores). The 24KE core uses separate I- and D-side watchpoints. So in the I-side watchpoints you'll find that `WatchLo0[R]` and `WatchLo0[W]` is fixed to zero, and so is `WatchLo2[I]`.

`M`: the `WatchHi[M]` bit is set whenever there is one more watchpoint register pair to find; your software should use it to figure out how many watchpoints there are (and should not rely on this manual for this purpose).

`G, ASID`: `WatchHi[ASID]` matches addresses from a particular address space (the "ASID" is like that in TLB entries) - except that you can set `WatchHi[G]` ("global") to match the address in any address space.

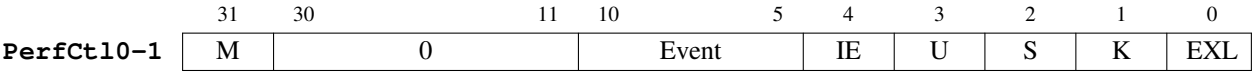
`Mask`: implements address ranges. Set bits in `WatchHi[Mask]` to mark corresponding `VAddr` address bits to be ignored when deciding whether this is a match.

`WatchHi[I], WatchHi[R], WatchHi[W]`: read `WatchHi` after a watch exception, and these fields tell you what type of access (if anything) matched.

5.4 Performance counters

The control register layout is shown in Figure 5-13.

Figure 5-13 Fields in the PerfCtl register



There are usually two counters, but software should check using the `PerfCtl[M]` bit (which indicates "at least one more").

Then the fields are:

`Event`: determines which event to count; see Table 5-2 below.

`IE`: set to cause an interrupt when the counter "overflows" into its bit31. 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.

`U, S, K, EXL`: count events in User mode, Supervisor mode, Kernel mode and Exception mode (ie when `Status[EXL]` is set) respectively. Set multiple bits to count in all cases.

The events which can be counted in the 24KE core are in Table 5-2. Blank fields are reserved.

Table 5-2 Performance counters 0 & 1 - events you can count

Event No	*Counter 0*	*Counter 1*
0	Cycles	
1	Instructions completed	
2	Branch instructions launched (whether completed or mispredicted)	Branch mispredictions
3	**`jr r31`** (return) instructions launched, whether completed or mispredicted	**`jr r31`** (return) mispredictions
4	**`jr`** (not r31) issues, which cost the same as a mispredict.	
5	Instruction micro-TLB accesses	Instruction micro-TLB misses
6	Data micro-TLB accesses	Data micro-TLB misses
7	Joint TLB instruction accesses	Joint TLB instruction misses
8	Joint TLB data (non-instruction) accesses	Joint TLB data (non-instruction) misses
9	Instruction cache accesses	Instruction cache misses
10	Data cache accesses	Data cache writebacks
11	Data cache misses	
12		
13		
14	Integer instructions completed	FPU instructions completed (not including loads/stores)
15	Loads completed (including FP)	Stores completed (including FP)
16	**`j/jal`** instructions completed	MIPS16 instructions completed
17	no-ops completed, ie instructions writing **`$0`**	integer multiply/divide unit instructions completed
18	Stalls	‘‘replay traps’’ (other than micro-TLB related)
19	**`sc`** instructions completed	**`sc`** instructions completed, but store failed (because the link bit had been cleared).
20	Prefetch instructions completed	‘‘superfluous’’ prefetch instructions (data was already in cache).
21	L2 cache writebacks	L2 cache accesses
22		
23	Exceptions taken	
24	‘‘cache fixup’’ events (specific to the 24KE family microarchitecture).	

Programming the 24KE™ core in user mode

Sections include:

- Section 6.1 “The multiplier”: multiply, multiply/accumulate and divide timings.
- Section 6.2 “Hardware registers”
- Section 6.3 “Prefetching data”: how it works.
- Section 6.4 “Tuning software for the 24KE family pipeline”: a brief guide[20].

6.1 The multiplier

As is traditional with MIPS CPUs, the integer multiplier is a semi-detached unit with its own pipeline. 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`.

None of the operations ever produce an exception - even divide-by-zero is silent - most compilers insert explicit check code.

The 24KE core multiplier is a high performance one; multiply/accumulate instructions can run at a rate of 1 per clock, but a 32×32 3-operand multiply takes five clocks longer than a simple ALU operation. Divides use a bit-per-clock algorithm, which is short-cut for smaller dividends. The result is that many of these operations will not be finished in time for the next instruction to proceed without delay; the following table shows the “maximum delay” - that’s the number of clocks the CPU will have to wait when a result is produced and consumed by consecutive instructions. The delays are summarized in Table 6-3, found in Section 6.4.4 “Data dependency delays classified”.

6.2 Hardware registers

The 24KE 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. See Section 9.1.2 “Release 2 of the MIPS32® Architecture - Hardware registers from user mode”.

6.3 Prefetching data

MIPS32 CPUs are also used for “media” computations where you might once have needed a DSP. These computations often feature loops accessing large arrays, and the run-time is often dominated by cache misses.

These are excellent candidates for using the **`pref`** instruction, which gets data into the cache without affecting the CPUs other state. In a well-optimized loop with prefetch, data for the next iteration can be fetched into the cache in parallel with computation for the last iteration.

It’s a pretty major principle that **`pref`** should have *no software-visible effect* other than to make things go faster. **`pref`** is logically a no-op[21].

The **`pref`** instruction comes with various possible “hints” which allow the program to express its best guess about the likely fate of the cache line.

The 24KE core acts on hints as summarized in Table 6-1.

[20] Should grow examples as they become available.

[21] This isn’t quite true any more; **`pref`** with the “PrepareForStore” hint can zero out some data which wasn’t previously zero.

Table 6-1 Hints for 'pref' instructions

No	*Hint Name*	*What happens in the 24KE core*	*Why would you use it?*
0 1	load store†	Read the cache line into the D-cache if not present.	When you expect to read the data soon. Use ''store'' hint if you also expect to modify it.
4 5	load_streamed store_streamed†	Fetch data, but always use cache way zero - so a large sequence of ''streamed'' prefetches will only ever use a quarter of the cache.	For data you expect to process sequentially, and can afford to discard from the cache once processed
6 7	load_retained store_retained†	Fetch data, but never use cache way zero. That means if you do a mixture of ''streamed'' and ''retained'' operations, they will not displace each other from the cache.	For data you expect to use more than once, and which may be subject to competition from ''streamed'' data.
25	writeback_invalidate/ nudge	If the line is in the cache, invalidate it (writing it back first if it was dirty). Otherwise do nothing. However (with the 24KE 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.

† Corresponding load/store prefetch hints act the same on the 24KE CPU. But it's good practice to use a ''store'' hint for any data you plan to change, for portability to cache-coherent systems which distinguish exclusive and shared ownership of cached data - in such a system the ''store'' hint will obtain an exclusive, writable, copy of the data.

6.4 Tuning software for the 24KE 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 24KE core is a pipelined design, and the pipeline and some of its consequences are described in Section 1.2 "A brief guide to the 24KE™ core implementation". That leads to a class of possible delays to do with data dependencies. 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 "big-ticket" items: cycles lost to cache misses and branches.

6.4.1 Cache delays and mitigating their effect

In a typical 24KE 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 50-100 instructions. A miss or uncached read (perhaps of a device register) may easily be several times slower. These really are big-ticket items!

In fact, because these delays are so large, there's not a lot you can do. But every little helps:

- The 24KE 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 early.

Any loop which walks predictably through a large array is a candidate for prefetch.

The **`pref PrepareForStore`** prefetch saves a cache refill read, for cache lines which you intend to overwrite in their entirety. Read more about prefetch in Section 6.3 "Prefetching data".

Tuning data-intensive common functions

Bulk operations like `bcopy()` and `bzero()` will benefit from some 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.

6.4.2 Branch delay slot

It's a feature of the MIPS architecture that it always attempts to execute the instruction immediately following a branch. The rationale for this is that it's extremely difficult to fetch the branch target quickly enough to avoid a delay, so the extra instruction runs "for free"...

Most of the time, the compiler deals well with this single delay slot. MIPS low-level programmers find it odd at first, but you get used to it!

6.4.3 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 1.2.1 "Branches and branch delays" 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[22] instructions (officially deprecated by the MIPS32 architecture because they may perform poorly on more sophisticated or wider-issue hardware) are predicted just like any other branch.

Although deprecated, the branch-likely instructions will probably improve the performance of loops where there is no other way of avoiding a no-op in a loop-closing branch's delay slot. If you're tempted to use this, we strongly recommend you make the code conditional on a `#define` variable tied specifically to the 24KE family.

6.4.4 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 just too complicated).

In fact, we won't set out the tables exactly like that. The MIPS instruction set is efficient because, most of the time, dependent instructions can be run nose-to-tail without delay. For all registers, there is a "standard" place in the pipeline where the producer should deliver its value and another place in the pipeline where the consumer picks it up[23]. Producer/consumer delays happen when either the producer is late delivering a result to the register (we'll abbreviate to "lazy"), or the consumer insists on obtaining its operand early (we'll abbreviate to "eager"). Of course, both may happen: in that case the delays add up.

It's important to be clear what class of registers is involved in any of these delays. For non-floating-point user-level code, there are just three classes of registers to consider:

- General purpose registers ("GPR");
- The `hi`/`lo` pair together with the three additional accumulators defined by the MIP DSP ASE ("ACC");
- The fields of the `DSPControl` register.

So that gives us two tables.

[22] The "likely" in the instruction name is historical, and pretty misleading.

[23] These are brought closer together by the magic of register file bypasses, but we don't need to get into the details here.

Delays caused by "eager consumers" reading values early

Table 6-2 Register → eager consumer delays

Reg	→	*Eager consumer*	*Del*	*Applies when ...*
GPR	→	load/store	1	the GPR value is an address operand (store data is not needed early).
ACC	→	multiply instructions	1	the ACC value came from any *non-multiply* or *multiply instructions which saturate the accumulator value* (values generated by other multiply instructions are made available early, and thus avoid this delay).
ACC	→	DSP instructions which extract selected bits from an accumulator: **`extp`**..., **`extr`**... etc. DSP instructions which write a shifted value back to the accumulator: **`mthlip`**, **`shilo`**, **`shilov`**.	3	

Delays caused by "lazy producers" delivering values late

Table 6-3 Lazy producer → register delays

Lazy producer	→	*Reg*	*Del*	*Applies when ...*
Load	→	GPR	1	Always (familiar as the "load delay slot").
Multiply unit instructions producing a GPR result. Instructions reading accumulators and writing GPR (eg **`mflo`**).	→	GPR	4	Always (because the multiply unit pipeline is longer than the integer unit's).
DSP "ALU" instructions (which neither read nor write an accumulator, nor do a multiplication).	→	GPR	1	
Divide instruction	→	ACC	7 9 15 17 23 25 31 33	8-bit dividend 8-bit dividend & negative operand to **`div`** 16-bit dividend 16-bit dividend & negative operand to **`div`** 24-bit dividend 24-bit dividend & negative operand to **`div`** full-size dividend full-size dividend & negative operand to **`div`**

How to use the tables

Suppose we've got an instruction sequence like this one:

```
addiu       $a0, $a0, 8
lw          $t0, 0($a0)      # [1]
lw          $t1, 4($a0)
addu        $t2, $t0, $t1    # [2]
mul         $v0, $t2, $t3
sw          $v0, 0($a1)      # [3]
```

Then a look at the tables should help us discover whether any instructions will be held up. Look at the dependencies where an instruction is dependent on its predecessor:

[1] The **lw** will be held up by one clock, because its GPR address operand `$a0` was computed by the immediately preceding instruction (see the first box of Table 6-2. The second **lw** will be OK.

[2] The **addu** will be one clock late, because the load data from the preceding **lw** arrives late in the GPR `$t1` (see the first box of Table 6-3.)

[3] The **sw** will be 4 clocks late starting while it waits for a result from the multiply pipe (the second box of Table 6-3.)

These can be additive. In the pointer-chasing sequence:

```
lw          $t1, 0($t0)
lw          $t2, 0($t1)
```

The second load will be held up two clocks: one because of the late delivery of load data in `$t1` (first box of Table 6-3), plus another because that data is required to form the address (first box of Table 6-2.)

Missing delay information relating to DSPControl

Some DSP ASE instructions are dependent because they produce and consume values kept in fields of the `DSPControl` register. However, the most performance-critical of these dependencies are "by-passed" to make sure no delay will occur - those are the dependencies between:

addsc → `DSPControl[c]` → **addwc**
cmp.x → `DSPControl[ccond]` → **pick.x**
wrdsp → `DSPControl[pos,scount]` → **insv**

But other dependencies passed in `DSPControl` may cause delays; in particular the `DSPControl[ouflag]` bits set by various kinds of overflow are not ready for a succeeding **rddsp** instruction. The access is interlocked, and will lead to a delay of up to three clocks. We don't expect that to be a problem (but if you know different, please get in touch with MIPS Technologies).

More complicated dependencies

There can be delays which are dependent on the dynamic allocation of resources inside the CPU. In general you can't really figure out how much these matter by doing a static code analysis, and we earnestly advise you to get some kind of high-visibility cycle-accurate simulator or trace equipment (probably based on Section 5.2 "PDtrace™ instruction trace facility").

Advice on tuning DSP ASE instruction sequences

DSP algorithm functions are often the subject of intense tuning. There is more specific and helpful advice (with examples) included in the white paper [DSPWP] published by MIPS Technologies.

The MIPS32® DSP ASE

The MIPS DSP ASE is provided to accelerate a large range of DSP algorithms. You can get most programming information from this chapter, but there's a great deal more detail in [MIPSDSP].

It's useful to summarize the range of target applications according to the data size (and therefore precision) typically used:

- *32-bit data*: audio (non-hand-held) decoding/encoding - a wide range of "hi-fi" standards for consumer audio or television sound.

  Raw audio data (as found on CD) is 16-bit; but if you do your processing in 16 bits you lose precision beyond acceptability for hi-fi.

- *16-bit data*: digital voice for telephony. International telephony code/decode standards include G.723.1 (8Ksample/s, 5-6Kbit/s data rate, 37ms delay), G.729 (8Kbit/s, 15ms delay) and G.726 (16-40Kbit/s, computationally simpler and higher quality, good for carrying analogue modem tones). Application-specific filters are used for echo cancellation, noise cancellation, and channel equalization.

  Also used for soft modems and much general "DSP" work (filters, correlation, convolution); lo-fi devices use 16 bits for audio.

- *8-bit data*: processing of printer images, JPEG (still) images and video data.

7.1 Features provided by the MIPS® DSP ASE

Those target applications can benefit from unconventional architecture features because they rely on:

- *Fixed-point fractional data types*: It is not yet economical (in terms of either chip size or power budget) to use floating point calculations in these contexts. DSP applications use fixed-point fractions. Such a fraction is just a signed integer, but understood to represent that integer divided by some power of two. A 32-bit fractional format where the implicit divisor is 2^{16} (65536) would be referred to as a Q15.16 format; that's because there are 16 bits devoted to fractional precision and 15 bits to the whole number range (the highest bit does duty as a sign bit and isn't counted).

  With this notation Q31.0 is a conventional signed integer, and Q0.31 is a fraction representing numbers between -1 and 1 (well, nearly 1). It turns out that Q0.31 is the most popular 32-bit format for DSP applications, since it won't overflow when multiplied (except in the corner case where -1×-1 leads to the just-too-large value 1). Q0.31 is often abbreviated to Q31.

  The DSP ASE provides support for Q31 and Q15 (signed 16-bit) fractions.

- *Saturating arithmetic*: It's not practicable to build in overflow checks to DSP algorithms - they need to be too fast. Clever algorithms may be built to be overflow-proof; but not all can be. Often the least worst thing to do when a calculation overflows is to make the result the most positive or most negative representable value. Arithmetic which does that is called *saturating* - and quite a lot of operations in the DSP ASE saturate (in many cases there are saturating and non-saturating versions of what is otherwise the same instruction).

- *Multiplying fractions*: if you multiply two Q31 fractions by re-using a full-precision integer multiplier, then you'll get a 64-bit result which consists of a Q62 result with (in the very highest bit) a second copy of the sign bit. This is a bit peculiar, so it's more useful if you always do a left-shift-by-1 on this value, producing a Q63 format (a more natural way to use 64 bits). Q15 multiplies which generate a Q31 value have to do the shift-left too. That's what all the **`mulq`**... instructions do.

- *Rounding*: some fractional operations implicitly discard less significant bits. But you get a better approximation if you bump the result by one when the discarded bits represent more than a half of the value of a 1 in the new LS position. That's what we mean by *rounding* in this chapter.

- *Multiply-accumulate sequences with choice of four accumulators* : (with fixed-point types, sometimes saturating).

  The 24KE already has quite a slick integer multiply-accumulate operation, but it's not so efficient when used for fractional and saturating operations.

  The sequences are made more usable by having four 64-bit result/accumulator registers - (the old MIPS multiply divide unit has just one, accessible as the `hi`/`lo` registers). The new `ac0` is the old `hi`/`lo`, for backward compatibility.

- *Benefit from "SIMD" operations.* : Many DSP calculations are a good match for "Single Instruction Multiple Data" or *vector* operations, where the same arithmetic operation is applied in parallel to several sets of operands.

  In the MIPS DSP ASE, some operations are SIMD type - two 16-bit operations or four 8-bit operations are carried out in parallel on operands packed into a single 32-bit general-purpose register. Instructions operating on vectors can be recognized because the name includes **`.ph`** (paired-half, usually signed, often fractional) or **`.qb`** (quad-byte, always unsigned, only occasionally fractional).

The DSP ASE hardware involves an extensive re-work of the normal integer multiply/divide unit. As mentioned above it has four 64-bit accumulators (not just one) and a new control register, described immediately below.

7.2 The DSP ASE control register

This is a part of the user-mode programming model for the DSP ASE, and is a 32-bit value read and written with the **`rddsp`**/**`wrdsp`** instructions. It holds state information for some DSP sequences.

Figure 7-1 Fields in the DSPControl Register

	31 28	27 24	23 16	15	14	13	12 7	6	5 0
`DSPControl`	0	ccond	ouflag	0	EFI	c	scount	0	pos

In Figure 7-1:

`ccond`: condition bits set by compare instructions (there have to be four to report on compares between vector types). "Compare" operations on scalars or vectors of length two only touch the lower-numbered bits. `DSPControl` bits 31:28 are used for more `ccond` bits in 64-bit machines.

`ouflag`: one of these bits may be set when a result overflows (whether or not the result is saturated depends on the instruction - the flag is set in either case). The "ou" stands for "overflow/underflow" - "underflow" is used here for a value which is negative but with excessive absolute value.

Any overflowed/underflowed result produced by any DSP ASE instruction sets a `ouflag` bit, *except* for **`addsc`**/**`addwc`** and **`shilo`**/**`shilov`**.

The 6 bits are set according to the destination of the operation which overflowed, and the kind of operation it was:

Bit No	*Overflowed destination/instruction*
16-19	Destination register is a multiply unit accumulator: separate bits are respectively for accumulators 0-3.
20	Add/subtract.
21	Multiplication of some kind.
22	Shift left or conversion to smaller type
23	Accumulator shift-then-extract

`EFI`: set by any of the accumulator-to-register bitfield extract instructions **`extp`**, **`extpv`**, **`extpdp`**, or **`extpdp`**. It's set to 1 if and only if the instruction finds there are insufficient bits to extract. That is, if `DSPControl[pos]` - which is supposed to mark the highest-numbered bit of the field we're extracting - is less than the size value specified by the instruction.

`c`: Carry bit for 32-bit add/carry instructions **`addsc`** and **`addwc`**.

`scount, pos`: Fields for use by ‘‘variable’’ bitfield insert and extract instructions, such as **`insv`** (the normal MIPS32 **`ins`**/**`ext`** instructions have the field size and position hard-coded in the instruction).

`scount` specifies the size of the bit field to be inserted, while `pos` specifies the insert position.

Caution: in all inserts (following the lead of the standard MIPS32 insert/extract instructions) `pos` is set to the lowest bit number in the field. But in the DSP ASE extract-from-accumulator instructions (**`extp`**, **`extpv`**, **`extpdp`** and **`extpdpv`**), `pos` identifies the *highest*-numbered bit in the field.

The latter two (‘‘dp’’) instructions post-decrement `pos` (by the bitfield length `size`), to help software which is unpacking a series of bitfields from a dense data structure.

The **`mthlip`** instruction will increment the `pos` value by 32 after copying the value of `lo` to `hi`.

7.2.1 DSP accumulators

Whereas a standard MIPS32 architecture CPU has just one 64-bit multiply unit accumulator (accessible as `hi`/`lo`), the DSP ASE provides three 64-bit accumulators. Instructions accessing the extra accumulators specify a 2-bit field as 0-3 (0 selects the original accumulator).

7.3 Software detection of the DSP ASE

You can find out if your core supports the DSP ASE by testing the `Config3[DDSP]` bit (see Figure 2-3 ‘‘Fields in the Config3 register’’).

Then you need to enable the instruction set by setting `Status[MX]` to 1.

7.4 DSP instructions

The DSP instruction set is nothing like the regular and orthogonal MIPS32 instruction set. It's a collection of special-case instructions, in many cases aimed at the known hot-spots of important algorithms.

We'll summarize the instructions under headings, but then list all of them in Table 7-2 "DSP instructions in alphabetical order", an alphabetically-ordered list which provides a terse but usually-sufficient description of what each instruction does.

7.4.1 Hints in instruction names

An instruction's name may have some suffixes which are often informative:

q: generally means it treats operands as fractions (which isn't important for adds and subtracts, but is important for multiplications and convert operations);

_s: usually means the full-precision result is saturated to the size of the destination;

_sa: is used for instructions which saturate intermediate results before accumulating;

r: denotes rounding (see above);

.w, .ph, .qb: suggest the operation is dealing with 32-bit, paired-half or quad-byte values respectively. Where there are two of these (as in **macq_s.w.phl**) the first one suggests the type of the result, and the second the type of the operand(s).

v: (in a shift instruction) suggests that the shift amount is defined in a register, rather than being encoded in a field of the instruction.

To help you get your arms around this collection of instructions we'll group them by likely usage - guided by the type of the result performed, with an eye to the application. The multiplication instructions are more tricky: most of them have multiple uses. We've sorted them by the most obvious use (likely also the most common). The classes are:

- Arithmetic - 64-bit
- Arithmetic - saturating and/or SIMD Types
- Bit-shifts - saturating and/or SIMD types
- Comparison and "conditional-move" operations on SIMD types - includes **pick** instructions.
- Conversions to and from SIMD types
- Multiplication - SIMD types with result in GP register
- Multiply Q15s from paired-half and accumulate
- Load with register+register address
- DSPControl register access
- Accumulator access instructions
- Dot products and building blocks for complex multiplication - includes full-word (Q31) multiply-accumulate
- Other DSP ASE instructions - everything else...

7.4.2 Arithmetic - 64-bit

addsc/addwc generate and use a carry bit, for efficient 64-bit add.

7.4.3 Arithmetic - saturating and/or SIMD Types

- *32-bit signed saturating arithmetic* : **addq_s.w**, **subq_s.w** and **absq_s.w**.
- *Paired-half and quad-byte SIMD arithmetic* : perform the same operation simultaneously on both 16-bit halves or all four 8-bit bytes of a 32-bit register. The "**q**" in the instruction mnemonic for the PH operations here is cosmetic: Q15 and signed 16-bit integer add/subtract operations are bit-identical - Q15 only behaves very differently when converted or multiplied.

The paired half operations are: **addq.ph/addq_s.ph**, **subq.ph/subq_s.ph** and **absq_s.ph**.

The quad-byte operations (all unsigned) are: **addu.qb/addu_s.qb**, **subu.qb/subu_s.qb**.

- *Sum of quad-byte vector* : **raddu.w.qb** does an unsigned sum of the four bytes found in a register, zero extends the result and delivers it as a 32-bit value.

7.4.4 Bit-shifts - saturating and/or SIMD types

All shifts can either have a shift amount encoded in the instruction, or - indicated by a trailing "**v**" in the instruction name - provided as a register operand. PH and 32-bit shifts have optional forms which saturate the result.

- *32-bit signed shifts* : include a saturating version of shift left, **shll_s.w**; and an auto-rounded shift right (just the "arithmetic", sign-propagating form): **shra_r.w**. Recall from above that rounding can be imagined as pre-adding a half to the least significant surviving bit.
- *Paired-half and quad-byte SIMD shifts* : **shll.ph/shllv.ph/shll_s.ph/shllv_s** are as above. For PH only there's a shift-right-arithmetic instruction ("arithmetic" means it propagates the sign bit downward) **shra.ph**, which has a variant which rounds the result **shra_r.ph**.

  The quad-byte shifts are unsigned and don't round or saturate: **shll.qb/shllv.qb**, **shrl.qb/shrlv.qb**.

7.4.5 Comparison and "conditional-move" operations on SIMD types

The "**cmp**" operations simultaneously compare and set flags for two or four values packed in a vector (with equality, less-than and less-than-or-equal tests). For PH that's **cmp.eq.ph**, **cmp.lt.ph** and **cmp.le.ph**. The result is left in the two LS bits of `DSPControl[ccond]`.

For quad-byte values **cmpu.eq.qb**, **cmpu.lt.qb** and **cmpu.le.qb** simultaneously compare and set flags for four bytes in `DSPControl[ccond]` - the flag relating to the bytes found in the low-order bits of the source register is in the lowest-numbered bit (and so on). There's an alternative set of instructions **cmpgu.eq.qb**, **cmpgu.lt.qb** and **cmpgu.le.qb** which leave the 4-bit result in a specified general-purpose register.

pick.ph uses the two LS bits of `DSPControl[ccond]` (usually the outcome of a paired-half compare instruction, see above) to determine whether corresponding halves of the result should come from the first or second source register. Among other things, this can implement a paired-half conditional move. You can reverse the order of your conditional inputs to do a move dependent on the complementary condition, too.

pick.qb does the same for QB types, this time using four bits of `DSPControl[ccond]`.

7.4.6 Conversions to and from SIMD types

Conversion operations from larger to smaller fractional types have names which start "**precrq...**" for "precision reduction, fractional". Conversion operations from smaller to larger have names which start "**prece...**" for "precision expansion".

- *Form vector from high/low parts of two other paired-half values* : **packrl.ph** makes a paired-half vector from two half vectors, swapping the position of each sub-vector. It can be used to acquire a properly formed sub-vector from a non-aligned data stream.
- *One Q15 from a paired-half to a Q31 value* : **preceq.w.phl/preceq.w.phr** select respectively the "left" (high bit numbered) or "right" (low bit numbered) Q15 value from a paired-half register, and load it into the result register as a Q31 (that is, it's put in the high 16 bits and the low 15 bits are zeroed).
- *Two bytes from a quad-byte to paired-half* : **precequ.ph.qbl/precequ.ph.qbr** picks two bytes from either the "left" (high bit numbered) or "right" (low bit numbered) halves of a quad-byte value, and unpacks to a pair of Q15 fractions.

  precequ.ph.qbla does the same, except that it picks two "alternate" bytes from bits 31-24 and 15-8, while **precequ.ph.qbra** picks bytes from bits 23-16 and 7-0.

  Similar instructions without the **q** - **preceu.ph.qbl**, **preceu.ph.qbr**, **preceu.ph.qbla** and **preceu.ph.qbra** - work on the same register fields, but treat the quantities as integers, so the 16-bit results get their low bits set.

- *2×Q31 to a paired-half*: both operands and result are assumed to be signed fractions, so **precrq.ph.w** just takes the high halves of the two source operands and packs them into a paired-half; **precrq_rs.ph.w** rounds and saturates the results to Q15.
- *2×paired-half to quad-byte* : you need two source registers to provide four paired-half values, of course. This is a fractional operation, so it's the low bits of the 16-bit fractions which are discarded.

  precrq.qb.ph treats the paired-half operands as unsigned fractions, retaining just the 8 high bits of each 16-bit component.

  precrqu_s.qb.ph treats the paired-half operands as Q15 signed fractions and both rounds and saturates the result (in particular, a negative Q15 fraction produces a zero byte, since zero is the lowest representable quantity).
- *Replicate immediate or register value to paired-half*: in **repl.ph** the value to be replicated is a 10-bit signed immediate value (thats in the range $-512 \leq x \leq 511$) which is sign-extended to 16 bits, whereas in **replv.ph** the value - assumed to be already a Q15 value - is in a register.
- *Replicate single value to quad-byte* : there's both a register-to-register form **replv.qb** and an immediate form **repl.qb**.

7.4.7 Multiplication - SIMD types with result in GP register

When a multiply's destination is a general-purpose register, the operation is still done in the multiply unit, and you should expect it to overwrite the `hi`/`lo` registers (otherwise known as `ac0`.)

- *8-bit×16-bit 2-way SIMD multiplication* : **muleu_s.ph.qbl/muleu_s.ph.qbr** picks the "left" (high bit numbered) or "right" (low bit numbered) pair of byte values from one source register and a pair of 16-bit values from the other. Two unsigned integer multiplications are done at once, the results unsigned-saturated and delivered to the two 16-bit halves of the destination.

  The asymmetric use of the source operands is not a bit like a Q15 operation. But 8×16 multiplies are heavily used in imaging and video processing (JPEG image encode/decode, for example).
- *Paired-half SIMD multiplication* : **mulq_rs.ph** multiplies two Q15s at once and delivers it to a paired-half value i n a general-purpose register, with rounding and saturation.
- *Multiply half-PH operands to a Q31 result* : **muleq_s.w.phl/muleq_s.w.phr** pick the "left"/"right" Q15 value respectively from each operand, multiply and store a Q31 value.

  "Precision-doubling" multiplications like this *can* overflow, but only in the extreme case where you multiply -1×-1, and can't represent 1 exactly.

7.4.8 Multiply Q15s from paired-half and accumulate

maq_s.w.phl/maq_s.w.phr picks either the left/high or right/low Q15 value from each operand, multiplies them to Q31 and accumulates to a Q32.31 result. The multiply is saturated only when it's -1×-1.

maq_sa.w.phl/maq_sa.w.phr differ in that the final result is saturated to a Q31 value held in the low half of the accumulator (required by some ITU voice encoding standards).

7.4.9 Load with register+register address

Previously available only for floating point data[24]: **lwx** for 32-bit loads, **lhx** for 16-bit loads (sign-extended) and **lbux** for 8-bit loads, zero-extended.

7.4.10 DSPControl register access

wrdsp rs,mask sets `DSPControl` fields, but only those fields which are enabled by a 1 bit in the 6-bit mask.

rddsp reads `DSPControl` into a GPR; but again it takes a mask field. Bitfields in the GPR corresponding to `DSPControl` fields which are not enabled will be set all-zero.

[24] Well, an integer instruction is also included in the MIPS SmartMIPS™ ASE.

The mask bits tie up with fields like this:

Table 7-1 Mask bits for instructions accessing the DSPControl register

Mask Bit:	*DSPControl field*
0	pos
1	scount
2	c
3	ouflag
4	ccond
5	EFI

7.4.11 Accumulator access instructions

- *Historical instructions which now access new accumulators* : the familiar **mfhi/mflo/mthi/mtlo** instructions now take an optional extra accumulator-number parameter.
- *Shift and move to general register* : **extr.w/extr_r.w/extr_rs.w** gets a 32-bit field from an accumulator (starting at bit 0 up to 31) and puts the value in a general purpose register. At your option you can specify rounding and signed 32-bit saturation.

  extrv.w/extrv_r.w/extrv_rs.w do the same but specify the field's starting bit number with a register.
- *Extract bitfield from accumulator* : **extp/extpv** takes a bitfield (up to 32 bits) from an accumulator and moves it to a GPR. The length of the field can be an immediate value or from a register. The position of the field is determined by `DSPControl[pos]`, which holds the bit number of the most significant bit.

  extpdp/extpdpv do the same, but also auto-decrement `DSPControl[pos]` to the bit-number just below the field you extracted.
- *Accumulator rearrangement* : **shilo/shilov** has a *signed* shift value between -32 and +31, where positive numbers shift right, and negative ones shift left. The "v" version, as usual, takes the shift value from a register. The right shift is a "logical" type so the result is zero extended.
- *Fill accumulator pushing low half to high* : **mthlip** moves the low half of the accumulator to the high half, then writes the GPR value in the low half. Generally used to bring 32 more bits from a bitstream into the accumulator for parsing by the various **ext**... instructions.

7.4.12 Dot products and building blocks for complex multiplication

In 2-dimensional vector math (or in any doubled-up step of a multiply-accumulate sequence which has been optimized for 2-way SIMD) you're often interested in the *dot product* of two vectors:

```
v[0]*w[0] + v[1]*w[1]
```

In many cases you take the dot product of a series of vectors and add it up, too.

Some algorithms use complex numbers, represented by 2D vectors. Complex numbers use `i` to stand for "the square root of -1", and a vector `[a, b]` is interpreted as `a + ib` (mathematicians leave out the multiply sign and use single-letter variables, habits which would not be appreciated in C programming!) Complex multiplication just follows the rules of multiplying out sums, remembering that `i*i = -1`, so:

```
(a + ib)*(c + id) = (a*c - b*d) + i(a*d + b*c)
```

Or in vector format:

```
[a, b] * [c, d] = [a*c - b*d, a*d + b*c]
```

The first element of the result (the "real component") is like a dot product but with a subtraction, and the second (the "imaginary component") is like a dot product but with the vectors crossed.

- *Q15 dot product from paired-half, and accumulate* : **dpaq_s.w.ph** does a SIMD multiply of the Q15 halves of the operands, then adds the results and saturates to form a Q31 fraction, which is accumulated into a Q32.31

fraction in the accumulator.

dpsq_s.w.ph does the same but subtracts the dot product from the accumulator.

For the imaginary component of a complex multiply, first swap the Q15 numbers in one of the register operands with a **rot** (bit-rotate) instruction.

For the real component of a complex Q15 multiply, you have the difference-of-products instruction **mulsaq_s.w.ph**, which parallel-multiplies both Q15 halves of the PH operands, then computes the difference of the two results and leaves it in an accumulator in Q32.31 format (beware: this does not accumulate the result).

- *16-bit integer dot-product from paired-half, and accumulate* : **dpau.h.qbl/dpau.h.qbr** picks two QB values from each source register, parallel-multiplies the corresponding pairs to integer 16-bit values, adds them together and then adds the whole lot into an accumulator. **dpsu.h.qbl/dpsu.h.qbr** do the same sum-of-products, but the result is then subtracted from the accumulator. In both cases, note this is integer (not fractional) arithmetic.
- *Q31 saturated multiply-accumulate* : is the nearest thing you can get to a dot-product for Q31 values. **dpaq_sa.l.w** does a Q31 multiplication and saturates to produce a Q63 result, which is added to the accumulator and saturated again. **dpsq_sa.l.w** does the same, except that the multiply result is subtracted from the accumulator (again, useful for the real component of a complex number).

7.4.13 Other DSP ASE instructions

- *Branch on DSPControl field* : **bposge32** branches if `DSPControl[pos]` ≥ 32.

  Typically the test is for "is it time to load another 32 bits of data from the bitstream yet?".
- *Circular buffer index update* : **modsub** takes an operand which packs both a maximum index value and an index step, and uses it to decrement a "buffer index" by the step value, but arranging to step from zero to the provided maximum.
- *Bitfield insert with variable size/position* : **insv** is a bit-insert instruction. It acts like the MIPS32 standard instruction **ins** except that the position and size of the inserted field are specified not as immediates inside the instruction, but are obtained from `DSPControl[pos]` (which should be set to the lowest numbered bit of the field you want) and `DSPControl[scount]` respectively.
- *Bit-order reversal* : **bitrev** reverses the bits in the low 16 bits of the register. The high half of the destination is zero.

  The bit-reverse operation is a computationally crucial step in buffer management for FFT algorithms, and a 16-bit operation supports up to a 32K-point FFT, which is much more than enough. A full 32-bit reversal would be expensive and slow.

7.5 Macros and typedefs for DSP instructions

It's useful to be able to use fragments of C code to describe what some instructions do. To do that, we need to be able to refer to fractional types, saturation and vectors. Here are the definitions we're using[25]:

```
typedef long long int64;
typedef int int32;

/* accumulator type */
typedef signed long long q32_31;

typedef signed int q31;

#define MAX31 0x7FFFFFFF
#define MIN31 -(1<<31)
#define SAT31(x) (x > MAX31 ? MAX31: x < MIN31 ? MIN31: x)

typedef signed short q15;
#define MAX15 0x7FFF
#define MIN15 -(1<<15)
#define SAT15(x) (x > MAX15 ? MAX15: x < MIN15 ? MIN15: x)

typedef unsigned char u8;
#define MAXUBYTE 255
#define SATUBYTE(x) (x > MAXUBYTE ? MAXUBYTE: x < 0 ? 0: x)

/* fields in the vector types are specified by relative bit
   position, but C definitions are in memory order, so these
   definitions need to be endianness-dependent */

#ifdef BIG_ENDIAN
typedef struct{
  q15 h1, h0;
} ph;

typedef struct{
  u8 b3, b2, b1, b0;
} qb;
#else
typedef struct{
  q15 h0, h1;
} ph;

typedef struct{
  u8 b0, b1, b2, b3;
} qb;
#endif
```

[25] This page needs more work, and I hope it will be improved in a future version of the manual.

7.6 Almost Alphabetically-ordered table of DSP ASE instructions

Table 7-2 DSP instructions in alphabetical order

Instruction	*Description*
absq_s.w rd,rt	Q31/signed integer absolute value with saturation
addq.ph rd,rs,rt **addq_s.ph rd,rs,rt**	2×SIMD Q15 addition, without and with saturation of the result
addq_s.w rd,rs,rt	Q31/signed integer addition with saturation
addsc rd,rs,rt **addwc rd,rs,rt**	Add setting carry, then add with carry. The carry bit is kept in `DSPControl[c]`. So to add the 64-bit values in registers `yhi`/`ylo`, `zhi`/`zlo` to produce a 64-bit value in `xhi`/`xlo`, just do: `addsc xlo, ylo, zlo; addwc xhi, yhi, zhi`
addu.qb rd,rs,rt **addu_s.qb rd,rs,rt**	4×SIMD QBYTE addition, without and with SATUBYTE saturation.
bitrev rd,rt	Delivers the bit-reversal of the low 16 bits of the input (result has high half zero).
bposge32 offset	Branch if `DSPControl[pos]` >= 32. Like most branch instruction, it has a 16-bit "PC-relative" target encoding.
cmp.eq.ph rs,rt **cmp.le.ph rs,rt** **cmp.lt.ph rs,rt**	Signed compare of both halves of two paired-half ("PH") values. Results are written into `DSPControl[ccond1-0]` for high and low halves respectively (1 for true, 0 for false). A signed compare works for both Q15 or signed 16-bit values.
cmpgu.eq.qb rd,rs,rt **cmpgu.le.qb rd,rs,rt** **cmpgu.lt.qb rd,rs,rt**	Unsigned simultaneous compare of all four bytes in quad-byte values. The four result bits are written into the four LS bits of general register `rd`.
cmpu.eq.qb rs,rt **cmpu.le.qb rs,rt** **cmpu.lt.qb rs,rt**	Unsigned simultaneous compare of all four bytes in quad-byte values. The four result bits are written into register `DSPControl[cond3-0]`.
dpaq_s.w.ph ac,rs,rt	"Dot product and accumulate", with Q31 saturation of each multiply result: `ph rs,rt; ac += SAT31(rs.h0*rt.h0 + rs.h1*rt.h1);` The accumulator is effectively used as a Q32.31 fraction.
dpaq_sa.l.w ac,rs,rt	Q31 saturated multiply-accumulate

dpau.h.qbl
 	
dpau.h.qbr
	`qb rs, rt;`
	`ac += rs.b3*rt.b3 + rs.b2*rt.b2;`
	Dot-product and accumulate of quad-byte values ("l" for left, because these are the higher bit-numbered bytes in the 32-bit register).
	Not a fractional computation, just unsigned 8-bit integers.
	Then for the lower bit-numbered bytes:
	`qb rs, rt;`
	`ac += rs.b1*rt.b1 + rs.b0*rt.b0;`

dpsq_s.w.ph ac,rs,rt	Paired-half fractional "dot product and subtract from accumulator"
	`ph rs, rt;`
	`q32_31 ac;`
	`ac -= SAT31(rs.h1*rt.h1 + rs.h0*rt.h0);`

dpsq_sa.l.w ac,rs,rt	Q31 saturated fractional-multiply, then subtract from accumulator:
	`q31 rs, rt; q32_31 ac;`
	`ac -= SAT31(rs*rt);`

Table 7-2 DSP instructions in alphabetical order

Instruction	*Description*

dpsu.h.qbl ac,rs,rt
 	
dpsu.h.qbr ac,rs,rt
	QB format dot-product and subtract from accumulator. This is an integer (not fractional) multiplication and comes in "left" and "right" (higher/lower-bit numbered pair) versions:
	`qb rs,rt;`
	`ac -= rs.b3*rt.b3 + rs.b2*rt.b2;`
	`qb rs,rt;`
	`ac -= rs.b1*rt.b1 + rs.b0*rt.b0;`

extp rt,ac,size **extpdp rt,ac,size** **extpdpv rt,ac,rs** **extpv rt,ac,rs**	Extract bitfield from an accumulator to register. The length of the field (number of bits) can be an immediate constant or can be provided by a second source register (in the **v** variants). The field position, though, comes from `DSPControl[pos]`, which marks the highest-numbered bit of the field (note that the MIPS32 standard bitfield extract instructions specify the *lowest* bit number in the field). In the **dp** variants like **extpdp/extpdpv**, `DSPControl[pos]` is auto-decremented by the length of the field extracted, which is useful when unpacking the accumulator into a series of fields.

extr.w rt,ac,shift
extr_r.w rt,ac,shift
extr_rs.w rt,ac,shift
extrv.w rt,ac,rs
extrv_r.w rt,ac,rs
extrv_rs.w rt,ac,rs
	Extracts a bit field from an accumulator into a general purpose register. The LS bit of the extracted field can start anywhere from bit zero to 31 of the accumulator:
	`int64 ac; unsigned int rt;`
	`rt = (ac >> shift) & 0xFFFFFFFF;`
	At option you can specify rounding (**_r** names):
	`int64 ac; unsigned int rt;`
	`rt = ((ac + 1<<(shift-1)) >> shift) & 0xFFFFFFFF;`
	and signed 32-bit saturation of the result (**_s/_rs** names).
	The **extrv...** variants specify the shift amount (still limited to 31 positions) with a register.

extr_s.h rt,ac,shift **extrv_s.h rt,ac,rs**	Obtain a right-shifted value from an accumulator and form a signed 16-bit saturated result.
insv rt,rs	The bitfield insert in the standard MIPS32 instruction set is **ins rt,rs,pos,size**, and the position and size must be constants (encoded as immediates in the instruction itself). This instruction permits the position and size to be calculated by the program, and then supplied as `DSPControl[pos]` and `DSPControl[scount]` respectively. In this case `DSPControl[pos]` must be set to the *lowest* numbered bit in the field to be inserted: yes, that's different from the **extp...** instructions.
lbux rd,index(base) **lhx rd,index(base)** **lwx rd, index(base)**	Load operations with register+register address formation. **lbux** is a load byte and zero extend, **lhx** loads half-word and sign-extends, and **lwx** loads a whole word. The full address must be naturally aligned for the data type.
maq_s.w.phl ac,rs,rt **maq_s.w.phr ac,rs,rt** **maq_sa.w.phl ac,rs,rt** **maq_sa.w.phr ac,rs,rt**	Non-SIMD Q15 multiply-accumulate, with operands coming from either the "left" (higher bit number) or "right" (lower bit number) half of each of the operand registers. In all versions the Q15 multiplication is saturated to a Q31 results. The "_sa" variants saturates the add result in the accumulator to a Q31, too.

Table 7-2 DSP instructions in alphabetical order

Instruction	*Description*
mfhi rd, ac **mflo rd, ac**	Legacy instruction, which now works on new accumulators (if you provide a second nonzero argument). Copies high/low half (respectively) of accumulator to `rd`.

modsub rd,rs,rt	Circular buffer index update. `rt` packs both the decrement amount (low 8 bits) and the highest index (high 24 bits), then this instruction calculates:
	`rd = (rs == 0) ?`
	`((unsigned) rt >> 8): rs - (rt & 0xFF);`

mthi rs, ac	Legacy instruction working on new accumulators. Moves data from `rd` to the high half of an accumulator.
mthlip rs, ac	Moves the low half of the accumulator to the high half, then writes the GPR value in the low half.
mtlo rs, ac	Legacy instruction working on new accumulators. Moves data from `rd` to the low half of an accumulator.
muleq_s.w.phl rd,rs,rt **muleq_s.w.phr rd,rs,rt**	Multiply selected Q15 values from "left"/"right" (higher/lower numbered bits) of `rd`/`rs` to a Q31 result in a general purpose register, Q31-saturating. Like all multiplies which target general purpose registers, it may well use the multiply unit and overwrite `hi`/`lo`, also known as `ac0`.

muleu_s.ph.qbl rd,rs,rt
muleu_s.ph.qbr rd,rs,rt
	A 2×SIMD 16-bit×8-bit multiplication.
	muleu_s.ph.qbl does something like:
	`rd = ((LL_B(rs)*LEFT_H(rt)) << 16) |`
	`((LR_B(rs)*RIGHT_H(rt));`
	Note that the multiplications are unsigned integer multiplications, and each half of the result is unsigned-16-bit-saturated.
	The asymmetric source operands are quite unusual, and note this is not a fractional computation.
	muleu_s.ph.qbr is the same but picks the RL and RR (low bit numbered) byte values from `rs`.

mulq_rs.ph rd,rs,rt	2×SIMD Q15 multiplication to two Q15 results. Result in general purpose register, `hi`/`lo` or `ac0` may be overwritten.

mulsaq_s.w.ph ac,rs,rt	`ac += (LEFT_H(rs)*LEFT_H(rt)) -`
	`(RIGHT_H(rs)*RIGHT_H(rt));`
	The multiplications are done to Q31 values, saturated if they overflow (which is only possible when `-1×-1` makes `+1`). The accumulator is really a Q32.31 value, so is unlikely to overflow; no overflow check is done on the accumulation.

packrl.ph rd,rs,rt	pack a "right" and "left" half from different registers, ie `rd = (((rs & 0xFFFF) << 16)	(rt >> 16) & 0xFFFF);`

pick.ph rd,rs,rt	Like a 2-way SIMD conditional move:
	`ph rd,rs,rt;`
	`rd.l = DSPControl[ccond1] ? rs.l: rt.l;`
	`rd.r = DSPControl[ccond0] ? rs.r: rt.r;`

Table 7-2 DSP instructions in alphabetical order

Instruction	*Description*

pick.qb rd,rs,rt	Kind of a 4-way SIMD conditional move:
	`qb rd,rs,rt;`
	`rd.ll = DSPControl[ccond3] ? rs.ll: rt.ll;`
	`rd.lr = DSPControl[ccond2] ? rs.lr: rt.lr;`
	`rd.rl = DSPControl[ccond1] ? rs.rl: rt.rl;`
	`rd.rr = DSPControl[ccond0] ? rs.rr: rt.rr;`

preceq.w.phl rd,rt **preceq.w.phr rd,rt**	Convert a Q15 value (either left/high or right/low half of `rt`) to a Q31 value in `rd`.
precequ.ph.qbl rd,rt **precequ.ph.qbla rd,rt** **precequ.ph.qbr rd,rt** **precequ.ph.qbra rd,rt**	Simultaneously convert two unsigned 8-bit fractions from `rt` to Q15 and load into the two halves of `rd`. **precequ.ph.qbl** uses `rt.ll`/`rt.lr`; **precequ.ph.qbla** uses `rt.ll`/`rt.rl`; **precequ.ph.qbr** uses `rt.rl`/`rt.rr`; and **precequ.ph.qbra** uses `rt.lr`/`rt.rr`.
preceu.ph.qbl rd,rt **preceu.ph.qbla rd,rt** **preceu.ph.qbr rd,rt** **preceu.ph.qbra rd,rt**	Zero-extend two unsigned byte values from `rt` to unsigned 16-bit and load into the two halves of `rd`. **preceu.ph.qbl** uses `rt.ll`/`rt.lr`; **preceu.ph.qbla** uses `rt.ll`/`rt.rl`; **preceu.ph.qbr** uses `rt.rl`/`rt.rr`; and **preceu.ph.qbra** uses `rt.lr`/`rt.rr`.
precrq.ph.w rd,rs,rt **precrq_rs.ph.w rd,rs,rt**	**precrq.ph.w** makes a paired-Q15 value by taking the MS bits of the Q31 values in `rs` and `rt`, like this: `rd = (rs & 0xFFFF0000) | ((rt>>16) & 0xFFFF);` **precrq_rs.ph.w** is the same, but rounds and Q15-saturates both half-results.

precrq.qb.ph rd,rs,rt	Form a quad-byte value from two paired-halves. We use the upper 8 bits of each halfword value, as if we were converting an unsigned 16-bit fraction to an unsigned 8-bit fraction. In C:
	`rd = (rs & 0xFF000000) | (rs<<8 & 0xFF0000) |`
	`(rt>>16 & 0xFF00) | (rt>>8 & 0xFF);`

precrqu_s.qb.ph **precrqu_s.qb.ph rd,rs,rt**	Does the same, but each conversion is rounded and saturated to an unsigned byte. Note in particular that a negative Q15 quantity yields a zero byte, since zero is the smallest representable value.
raddu.w.qb rd,rs	Set `rd` to the unsigned 32-bit integer sum of the four unsigned bytes in `rs`.
rddsp rt,mask	Read the contents of the `DSPControl` register into `rt`, but zeroing out any fields for which the appropriate mask bit is zeroed, see Table 7-1 above.
repl.ph rd,imm **replv.ph rd,rt**	Replicate the same signed value into the two halves of a PH value in `rd`; the value is either provided as an immediate whose range is limited between -512 and +511 (**repl.ph**) or from the `rt` register (**replv.ph**).
repl.qb rd,imm **replv.qb rd,rt**	Replicate the same 8-bit value into all four parts of a QB value in `rd`; the value can come from an immediate constant, or the `rt` register of the **replv.qb** instruction.
shilo ac,shift **shilov ac,rs**	Do a right or left shift (use a negative value for a left shift) of a 64-bit accumulator. The right shift is "logical", bringing in zeroes into the high bits. **shilo** takes a constant shift amount, while **shilov** get the shift amount from `rs`. The shift amount may be no more than 31 right or 32 left.

Table 7-2 DSP instructions in alphabetical order

Instruction	*Description*
shll.ph rd, rt, sa **shllv.ph rd, rt, rs** **shll_s.ph rd, rt, sa** **shllv_s.ph rd, rt, rs**	2×SIMD (paired-half) shift left. The "**v**" versions take the shift amount from a register, and the "**_s**" versions saturate the result to a signed 16-bit range.
shll.qb rd, rt, sa **shllv.qb rd, rt, rs**	4×SIMD quad-byte shift left, with shift-amount-in-register and saturating (to an unsigned 8-bit result) versions.
shll_s.w rd, rt, sa **shllv_s.w rd, rt, rs**	Signed 32-bit shift left with saturation, with shift-amount-in-register **shllv_s** option.
shra.ph rd, rt, sa **shra_r.ph rd, rt, sa** **shrav.ph rd, rt, rs** **shrav_r.ph rd, rt, rs**	2×SIMD paired-half shift-right arithmetic ("arithmetic" because the vacated high bits of the value are replaced by copies of the input bit 16, the sign bit) - thus performing a correct division by a power of two of a signed number. As usual the **shra_v** variant has the shift amount specified in a register. The **_r** versions round the result first (see the bullet on rounding above).
shra_r.w rd, rt, sa **shrav_r.w rd, rt, rs**	32-bit signed/arithmetic shift right with rounding, see the bullet on rounding.
shrl.qb rd, rt, sa **shrlv.qb rd, rt, rs**	4×SIMD shift right logical ("logical" means that the vacated high bits are filled with zero, appropriate since the byte quantities in a quad-byte are usually treated as unsigned.)
subq.ph rd,rs,rt **subq_s.ph rd,rs,rt**	2×SIMD subtraction. **subq_s.ph** saturates its results to a signed 16-bit range.
subq_s.w rd,rs,rt	32-bit saturating subtraction.
subu.qb rd,rs,rt **subu_s.qb rd,rs,rt**	4×SIMD quad-byte subtraction. Since quad-bytes are treated as unsigned, the saturating variant **subu_s.qb** works to an unsigned byte range.
wrdsp rt,mask	Write the `DSPControl` register with data from `rt`, but leaving unchanged any fields for which the appropriate mask bit is zeroed, see Table 7-1 above.

7.7 DSP ASE instruction timing

Most DSP ASE operations are pipelined, and instructions can often be issued at the maximum CPU rate, but getting results back into the general-purpose register file takes a few clocks. The timings are generally fairly similar to those for the standard multiply instructions, and are listed - together with delays for the standard instruction set - in Section 6.4.4 "Data dependency delays classified".

Floating point unit

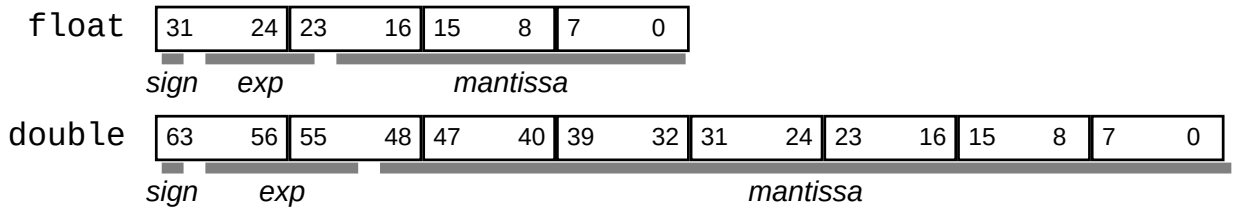
The 24KEf™ member of the 24KE family has a hardware floating point unit (FPU). This:

- *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 [MIPS64]; because the 24KE 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 Chapter 9 "What's new 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 24KE 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 8.4.2 "FPU "unimplemented" exceptions (and how to avoid them)".
- *Omits "paired single" and MIPS-3D extensions:* : those are primarily aimed at 3D graphics, and are described as optional in [MIPS64].
- *Uses an autonomous 7-stage pipeline* : all data transfers are interlocked, so the programmer is never aware of the pipeline. Compiler writers and daemon subroutine tuners do need to know: there's timing information in Section 8.5 "FPU pipeline and instruction timing".

8.1 Data representation

If you'd like to read up on floating point in general you might like to read [SEEMIPSRUN]. But it's probably useful to remind you (in Figure 8-1) what 32-bit and 64-bit floating point numbers on MIPS architecture CPUs look like.

Figure 8-1 How floating point numbers are stored in a register



Just to remind you:

- *sign* : FP numbers are positive numbers with a separate sign bit; "1" denotes a negative number.
- *mantissa* : represents a binary number. But this is a floating point number, so the units depend on:
- *exp* : the exponent.

When 32-bit data is held in a 64-bit register, the high 32 bits are don't care.

The MIPS Architecture's 32-bit and 64-bit floating point formats are compatible with the definitions of "single precision" and "double precision" in [IEEE754].

FP registers can also hold simple 2s-complement signed integers too, just like the same number held in the integer registers. That happens whenever you load integer data, or convert to an integer data type.

Floating point data in memory is endianness-dependent, in just the same way as integer data is; the higher bit-numbered bytes shown in Figure 8-1 "How floating point numbers are stored in a register" 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.

8.2 Basic instruction set

Whenever it makes sense to do so, FP instructions exist in a version for each data type. In assembler that's denoted by a suffix of:

.s	single-precision
.d	double-precision
.w	32-bit integer (''word'')
.l	64-bit integer

There's a good readable summary of the floating point instruction set in [SEEMIPSRUN], and you can find the fine technical details in [MIPS64].

As a one-minute guide: the FPU provides basic arithmetic (add, multiply, subtract, divide and square root). It's all register-to-register (like the integer unit). It's written ''destination first'' like integer instructions; sometimes that's unexpected in that **cvt.d.s** is a ''convert from single to double''. It has a set of multiply/add instructions which work on *four* registers: **madd a,b,c,d** does

```
a = c*d + b
```

as a single operation. There are a rich set of conversion operations. A bewildering variety of compare instructions record their results in any one of eight condition flags, and there are branch and conditional-move instructions which test those flags.

You won't find any higher-level functions: no exponential, log, sine or cosine. This is a RISC instruction set, you're expected to get library functions for those things.

8.3 Floating point loads and stores

FP data does not normally pass through the integer registers; the FPU has its own load and store instructions. The FPU is conceptually a replaceable tenant of coprocessor 1: while arithmetic FP operations get recognizable names like **add.d**, the load/store instructions will be found under names like **ldc1** in [MIPS64] and other formal documentation. In assembler code, you'll more often use mnemonics like **l.d** which you'll find will work just fine.

Because FP-intensive programs are often dealing with one- or two-dimensional arrays of values, the FPU gets special load/store instructions where the address is formed by adding two registers; they're called **ldxc1** etc. In assembler you just use the **l.d** mnemonic with an appropriate address syntax, and all will be well.

8.4 Setting up the FPU and the FPU control registers

There's a fair amount of state which you set up to change the way the FPU works; this is controlled by fields in the FPU control registers, described here.

8.4.1 IEEE options

[IEEE754] defines five classes of exceptional result. For each class the programmer can select whether to get an IEEE-defined ''exceptional result'' or to be interrupted. Exceptional results are sometimes just normal numbers but where precision has been lost, but also can be an *infinity* or *NaN* (''not-a-number'') value.

Control over the interrupt-or-not options is done through the `FCSR[Enable]` field (or more cleanly through `FENR`, the same control bits more conveniently presented); see Table 8-1 below.

It's overwhelmingly popular to keep `FENR` zero and thus never generate an IEEE exception; see Section 8.5 ''FPU pipeline and instruction timing'' for why this is a particularly good idea if you want the best performance.

8.4.2 FPU "unimplemented" exceptions (and how to avoid them)

It's a long-standing feature of the MIPS Architecture that FPU hardware need not support every corner-case of the IEEE standard. But to ensure proper IEEE compatibility to the software system, an FPU which can't manage to generate the correct value in every case must detect a combination of operation and operands it can't do right. It then takes an *unimplemented* exception, which the OS should catch and arrange to software-emulate the offending instruction.

The 24KE core 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[26].

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 Table 8-1 "FPU (co-processor 1) control registers"), or provide a software emulator and resign yourself to a small number of "unimplemented" exceptions.

8.4.3 FPU control register maps

There are five FP control registers:

Table 8-1 FPU (co-processor 1) control registers

Conventional Name	*CP1 ctrl reg num*	*Description*
`FCSR`	31	Extensive control register - the only FPU control register on historical MIPS CPUs. Contains *all* the control bits. But in practice some of them are more conveniently accessed through `FCCR`, `FEXR` and `FENR` below.
`FIR`	0	FP implementation register: read-only information about the capability of this FPU.

`FCCR`
`FEXR`
`FENR`
	25
	26
	28
		Convenient partial views of `FCSR` are better structured, and allow you to update fields without interfering with the operation of independent bits.
		`FCCR` has FP condition codes, `FEXR` contains IEEE exceptional-condition information (cause and flag bits) you read, and `FENR` is IEEE exceptional-condition enables you write.

The FP implementation (FIR) register

Figure 8-2 shows the fields in `FIR` and the read-only value they always have for 24KE family FPUs:

Figure 8-2 Fields in the FIR register

	31 … 25	24	23	22	21	20	19	18	17	16	15 … 8	7 … 0
FIR	*0*	FC	*0*	F64	L	W	3D	PS	D	S	Processor ID	Revision
the 24KE core value		1		1	1	1	0	0	1	1	0x96	*whatever*

The fields have the following meanings:

- *FC*: "full convert range": the hardware will complete *any* conversion operation without running out of bits and causing an "unimplemented" exception.
- *F64/L/W/D/S*: this is a 64-bit floating point unit and implements 64-bit integer ("L"), 32-bit integer ("W"), 64-bit FP double ("D") and 32-bit FP single ("S") operations.

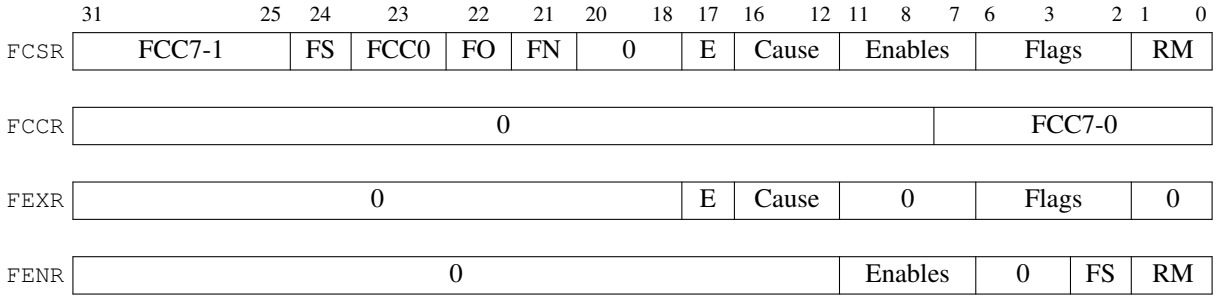
[26] IEEE754 defines an alternative "denormalized" representation for these numbers.

- *3D* : does not implement the MIPS-3D ASE.
- *PS* : does not implement the paired-single instructions described in [MIPS64]
- *Processor ID/Revision* : major and minor revisions of the FPU - as is usual with revisions it's very useful to print these out from a verbose sign-on message, and rarely a good idea to have software behave differently according to the values.

The FP control/status registers (FCSR, FCCR, FEXR, FENR)

Figure 8-3 shows all these registers and their bits

Figure 8-3 Floating point control/status register and alternate views



Where:

`FCC7-0`: the floating point condition codes: set by compare instructions, tested by appropriate branch and conditional move instructions.

`FS/FO/FN`: options to avoid "unimplemented" exceptions when handling tiny ("denormalized") numbers[27]. They do so at the cost of IEEE compatibility, by replacing the very small number with either zero or with the nearest nonzero quantity with a normalized representation.

The `FO` ("flush override") bit causes all tiny operand and result values to be replaced.

The `FS` ("flush to zero") bit causes all tiny operand and result values to be replaced, but additionally does the same substitution for any tiny intermediate value in a multiply-add instruction. This is provided both for legacy reasons, and in case you don't like the idea that the result of a multiply/add can change according to whether you use the fused instruction or a separate multiply and add.

The `FN` bit ("flush to nearest") bit causes all result values to be replaced with somewhat better accuracy than you usually get with `FS`: the result is either zero or a smallest-normalized-number, whichever is closer. Without FN set you can only replace your tiny number with a nonzero result if the "RP" or "RM" rounding modes (round towards more positive, round towards more negative) are in effect.

For full IEEE-compatibility you must set `FCSR[FS,FO,FN] == [0,0,0]`.

To get the best performance compatible with a guarantee of no "unimplemented" exceptions, set `FCSR[FS,FO,FN] == [1,1,1]`.

Just occasionally for legacy applications developed with older MIPS CPUs which did not have the `FO` and `FN` options, you might set `FCSR[FS,FO,FN] == [1,0,0]`.

`E`: (often shown in documents as part of the `Cause` array) is a status bit indicating that the last FP instruction caused an "unimplemented" exception, as discussed in Section 8.4.2 "FPU "unimplemented" exceptions (and how to avoid them)".

[27] 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[28] 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 (eg 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 bit 0 is zero.
1	Round toward zero - *RZ*
2	Round towards plus infinity - *RP* ‘‘Round up’’ (but unambiguous about what you do about negative numbers).
3	Round towards minus infinity - *RM*

8.5 FPU pipeline and instruction timing

The FPU has an autonomous pipeline which receives instructions from the integer pipeline. It most often runs at one half of the integer unit clock rate (it is configurable when your 24KE core is synthesized, but a full-core-speed FPU will substantially lower the frequency limit of the whole chip).

But don’t assume that half-speed is slow: this is an extremely powerful and capable FPU, enabling applications which could never have been built around a synthesizable core CPU[29] before.

The FP instruction continues down the integer pipeline, where nothing much happens to it. The timing headlines:

- FP instructions issued in the ‘‘wrong’’ half clock will be dealt with by stalling the whole integer pipe for a clock to get back in step. Tightly-tuned sequences of mixed instructions should aim to issue FP instructions every second, fourth or sixth instruction etc.
- Single-precision simple operations (multiply, add and multiply/add) deliver their result in 4 FPU clocks and you can start a new one every FPU clock.
- Double-precision simple operations deliver a result in five FPU clocks, and you can start a new one every two FPU clocks.
- Repeat rates may be slower when using the ‘‘legacy mode’’ (ie with `FCSR[FR]` set zero, where you get only 16 arithmetic registers).

[28] 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.

[29] Well, maybe they *could* if they had used MIPS Technologies 5Kf™ core...

- The MIPS architecture requires FP exceptions to be precise, which means any exception must be signalled before the next instruction in sequence has done anything irrevocable[30]. If an FP instruction might deliver a MIPS exception for any reason, it must be noted before the ‘‘runt’’ FP instruction in the integer pipe reaches the integer ‘‘MS’’ stage. If the FPU isn’t ready to confirm ‘‘no exception’’ by that point, the CPU must stall and wait for it - it can’t complete the instruction following the FPU instruction.

  I hope more details will follow in a later edition of this manual: but the bottom line is that if you configure your FPU in non-IEEE mode so it will never take an ‘‘unimplemented’’ exception, *and* leave `FCSR[Enables]` zero so it never takes an IEEE-condition exception, your CPU will not have to hang about waiting for FP exceptions to be resolved: your program will run faster.

- Some *really* simple operations run quicker, and more complex operations like divide and square-root take much longer, and typically occupy the whole FPU while they’re doing so - see Table 8-2.

Table 8-2 Floating point instruction timings - all in FPU clock periods

Instructions	*Latency*	*Repeat Rate*
Compare instruction (latency is to dependent branch or conditional move)	1-2	1
`mtc1`, **`mfc1`** (move between integer and FPU registers)	2	1
Floating point loads	3	1
`abs`, **`neg`**, all moves between FP registers, including conditional moves.	4	1
`cvt.d.s` (single to double)	4	1
All integer → FP conversions	4	1
Single-precision adds, multiplies and multiply-add	4	1
All FP → integer conversions (including **`ceil`** etc)	5	1
Double-precision adds, multiplies and multiply-add	5	2
`cvt.s.d` (double to single)	6	1
`recip.s`	13	10
Single-precision divide and **`sqrt`**	17	14
`rsqrt.s`	17	14
`recip.d`	25	21
Double-precision divide and **`sqrt`**	32	29
`rsqrt.d`	35	31

[30] That’s slightly oversimplified. You’ll probably understand it better by reading something like [SEEMIPSRUN].

What's new in Release 2 of the MIPS32® Architecture?

Release 2 of the MIPS32 specification was released in late 2002, and the first core conformant to it was MIPS Technologies 4Ke™. The 24KE core is the first family to be designed from the outset to MIPS32 Release 2.

9.1 Changes visible at user level

The 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.

9.1.1 Release 2 of the MIPS32® Architecture - new instructions for user-mode

The following instructions are new with the MIPS32 release 2 update:

Table 9-1 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`**.

9.1.2 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[31]; 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.
- *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).

[31] Strictly, it's the lesser of the I-cache and D-cache line size, but it's most unusual to make them different.

9.2 New privileged instructions

Table 9-2 details the (few) new privileged instructions:

Table 9-2 Release 2 of the MIPS32® Architecture - New privileged instructions

Instruction(s)	*Description*
`di rt` `ei rt`	Atomic disable/enable interrupt - clears/sets (respectively) the `Status[IE]` interrupt enable flag, but does so in one instruction. Moreover, the old `Status` value is stored in a GP register, so it can be reinstated with a **`mtc0`** instruction.
`rdpgpr rd, rt` `wrpgpr rd, rt`	Read/write "shadow" registers. The shadow set accessed is the one whose number is currently in `SRSCtl[PSS]`. Defaults to a register-to-register move if the previous and current register sets are the same (as they always are for a CPU without extra registers). See Section 9.6 "Shadow registers".

9.3 Hazard barriers - no more CPU-dependent no-ops

In any pipelined CPU, code sequences which change machine state through privileged operations or CP0 registers can produce strange behavior when an effect is deferred out of its normal instruction sequence (typically because the relevant control register only gets written some way down the pipe).

Traditionally, MIPS CPUs have 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 around with **`nop`** or **`ssnop`** instructions.

From Release 2 of the MIPS32 specification, though, this is replaced by *hazard barrier* instructions as shown in the table above: **`eret`**, **`jr.hb`** and **`jalr.hb`** are barriers to all side-effects, while **`ehb`** (a kind of nop-on-steroids) deals with all side effects except those affecting instruction fetch.

Hazard barrier instructions are likely to be implemented with a pipeline flush or something like it, so are quite costly: they should not be used indiscriminately. For efficiency you should use **`ehb`** where it is enough, and - where it's safe to do so - place the barrier instruction a few instructions after the side-effect causing instruction.

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 Release 2 CPUs.
- *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.

9.4 Exception entry points - a review

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, 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.

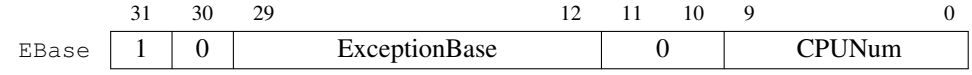
In MIPS CPUs prior to the MIPS32 architecture (with a few infrequent special cases) only common TLB miss exceptions got their own entry point; interrupts and all other OS-handled exceptions were all funneled through a single "general" exception entry point.

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]`.

Release 2: relocate all the exception entry points with EBase

Figure 9-1 Fields in the EBase register



With `EBase[ExceptionBase]` you can relocate *all* the RAM exception entry points (as a group) from their traditional starting point at `0x8000.0000` to any 4Kbyte[32] aligned address in kseg0 or kseg1 (that is any suitably aligned virtual address from `0x8000.0000`–`0xBFFF.F000`[33]). The ability to relocate exception entry points is vital for shared-memory multiprocessors which need separate exception entry points for each CPU, but may also be useful for applications which must juggle the memory map and find it inconvenient to have high-performance program memory from physical address zero.

The new exception base address is read as the bits 31–12 of the `EBase` register. Note that the `EBase[ExceptionBase]` field is defined to exclude bits 31–30, which are fixed to `10` to restrict the exception reassignment - a software definition of the field might more conveniently include the top bits, though they can't be changed.

[32] When you use vectored interrupts, the individual interrupt's entry point offset is OR'd with the interrupt base address, not added; so if you have a large number of widely-spaced interrupt entry points, you need more zeroes at the bottom of `EBase`.

For the (limiting) case: if you had the maximum 64 interrupt entry points with the maximum permitted (512 byte) spacing, your `EBase` should be 32Kbyte-aligned.

[33] Well, almost. The cache error exception entry point must really be in uncached memory; if you set `EBase` as if to move it into kseg0, in fact it will map to the corresponding kseg1 address. Equivalently, the cache error entry point moves with all the other entry points in physical memory: it's just always executed uncached.

EBase[CPUNum] is used for a different purpose = to return the read-only CPU number by which the various CPUs in a multiprocessor setup can identify themselves. Strictly, [MIPS32] requires that bits 11 and 10 be written only as zero.

9.4.1 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 9-3. You need to accept that BASE is 0x8000.0000 for CPUs without an EBase register (or where the software, ignoring the EBase register, leaves it at its power-on value); and that otherwise BASE is the 4Kbyte-aligned address found in EBase after you ignore the low 12 bits...

Table 9-3 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	0xBFC0.0480	EJTAG debug, when using normal ROM memory.
	0xBFC0.0000	Post-reset and NMI entry point.
ROM entry points (when Status[BEV]==1)	0xBFC0.0200	Simple TLB Refill (Status[EXL]==0).
	0xBFC0.0300	Cache Error
	0xBFC0.0400	Interrupt special (Cause[IV]==1).
	0xBFC0.0380	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 9.5 "Release 2 - enhanced interrupt system(s)".
	BASE+0x180	All others

9.5 Release 2 - enhanced interrupt system(s)

Release 2 brings significant new features for handling interrupts:

- 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 9.6 "Shadow registers". Interrupt handlers using shadow registers avoid the overhead of saving and restoring user GPR values.

- New readable `Cause[TI]` and `Cause[PCI]` bits 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 9-2.

Figure 9-2 Fields in the IntCtl register

	31 29	29 26	25 10	9 5	4 0
IntCtl	IPTI Int sig shared by timer	IPPCI Int input shared by perf counter	0	VS Vector spacing	0

Notes:

`IntCtl[IPTI,IPPCI]`: are read-only fields, telling you how the hardware is configured in its non-vectored and simple-vectored ("VI") interrupt modes. Each is a 3-bit binary number identifying which CPU interrupt input is shared by the internal timer interrupt (`IPTI`) or the performance counter overflow interrupt (`IPPCI`).

The interrupt is specified by giving the number of the `Cause[IP nn]` where the resulting interrupt is seen. Because `Cause[IP0-1]` are software interrupt bits, unconnected to any input, legal values for `IntCtl[IPTI]` and `IntCtl[IPPCI]` are between 2 and 7.

`IntCtl[VS]`: is writable to give you software control of the vector spacing; the spacing you get between consecutive entries is `IntCtl[VS]`×32 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 does give a zero spacing, so all interrupts arrive at the same address.

9.5.1 VI mode - interrupt signalling and priority

Most MIPS CPUs to date have a number of interrupt inputs which are simply bit-masked and ORed together to determine whether to take an interrupt exception; control transfers to the general exception entry point (used for everything except user TLB miss) and the `Cause[ExcCode]` field is set to show an interrupt. Software decides which interrupt source is responsible (and if there is more than one active and unmasked, which gets attention first).

Using a single entry point 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 from Release 2 of the MIPS32 architecture, if:

1. One of `Config3.VInt` and `Config3.EIC` is set (to indicate that your core has the vectored-interrupts feature); and:

2. you set `Cause[IV]` to request that interrupts use the special interrupt entry point; and:
3. You set `IntCtl[VS]` to set the spacing between successive interrupt entry points.

Then interrupt exceptions will go to multiple distinct entry points based either on six independent interrupts plus two software interrupts (`Config3.Vint` set) or - with an EIC-compatible interrupt controller, using the six lines to encode one of 63 possible interrupts, up to 63 different entry points (`Config3.EIC` set).

In VI mode, faced with six input signals and two software interrupts (that's the state of the read/write `Cause[IP1-0]` bits), the CPU prioritizes hardware interrupts first, and higher-numbered interrupts first, to produce a "vector number" in the range 0-7.

The vector number is multiplied by the "spacing" implied by the OS-written field `IntCtl.VS` to generate an offset. The spacing can be any power of two between 32 and 512 bytes. 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.

A wrinkle: MIPS CPUs have two internal interrupt sources; the counter/timer and (if implemented) performance counter overflow. These are generated as outputs at the core interface, and it's up to the system designer how to feed them back in. However, if each interrupt corresponds to a separate interface signal (always the case prior to the new "EIC mode" described in the next section) then they will each be signalled - either alone or OR'd with some other interrupt source - on one of the six interrupt inputs. In this case, the system designer should wire the preset fields `IntCtl[IPTI]` and `IntCtl[IPPCI]` to identify where the signals are attached; and the software should read those fields to find out.

9.5.2 External Interrupt Controller (EIC) mode

Embedded systems have lots of interrupts, typically far exceeding the six input signals available. Most systems have an external interrupt controller to allow these interrupts to be masked and selected. If your interrupt controller is EIC compatible, then it encodes its highest active interrupt request on the six input signals - now redefined as a bus with 64 possible values[34].

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. In the normal application or routine-kernel state no interrupt is in suspension, and `Status[IPL]` is likely to be zero; the CPU takes an interrupt exception when the core is willing to be interrupted[35] and the interrupt controller presents a number higher than the current value of `Status[IPL]` on its "bus".

Figure 9-3 Fields in the Cause register

	31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
Cause	BD	TI	CE	DC	PCI	0	IV	WP	0	IP7-2	IP1-0	0	Exc Code	0
									In EIC (external int ctrl) mode	RIPL				

As before, the interrupt handler will see the interrupt request number in `Cause[IP]` bits - see Figure 9-3; 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 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[36].

[34] 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).

[35] To take an interrupt, of course, `Status[IE]` must be set and `Status[EXL]` and `Status[ERL]` clear.

[36] 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.

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 defined in `IntCtl[RS]` and added to the special interrupt entry point, as described in the previous section.

The software interrupts are still usable; they are writable in `Cause` and the post-mask values are delivered to the external interrupt controller, which can deal with them in some suitable manner.

`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.

9.6 Shadow registers

In hardware terms, shadow registers are deceptively simple: just double or quadruple the size of the register file, and then add some more index bits to every register access: and you've got more registers. 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 GP registers. The CPU only changes register sets on an exception (details to follow) or when returning from an exception with **`eret`**.

Selecting shadow sets - SRSCtl

The architecture permits up to 16 shadow sets, but 2-4 is likely to be common. The set selectors are in the `SRSCtl` register, shown in Figure 9-4.

Figure 9-4 Fields in the SRSCtl register (shadow register set control)

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0	HSS	0	EICSS	0	ESS	0	PSS	0	CSS

In `SRSCtl`:

`SRSCtl[HSS]`: the number of available register sets minus one (ie the highest-numbered valid register set on this CPU). Read-only.

`SRSCtl[CSS]`: the register set currently in use. It's read-only here; set on any exception, replaced by the value in `SRSCtl[PSS]` on an **`eret`**.

`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).

`SRSCtl[PSS]`: the ''previous'' register set, which will be used following the next **`eret`**.

You can get at the values of registers in this set using **`rdpgpr`** and **`wrpgpr`**.

`SRSCtl[PSS]` is writable, allowing the OS to dispatch code in a new register set; load this value and then execute an **`eret`**.

`SRSCtl[EICSS]`: will be explained in the next section.

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 is used to fix 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.

If you are using neither VI nor EIC mode, the new set to be used will be taken from the default any-other-exception field `SRSCtl[ESS]`.

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 ''free'', no-need-to-save temporaries; they can be used to hold low-cost context for the interrupt routine which uses them.

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!

9.7 FPU changes in Release 2 of the MIPS32® Architecture

The main change is that a 32-bit CPU (like the 24KE core) can now be paired with a 64-bit floating point unit. The FPU itself is compatible with the description in [MIPS64].

The only new feature of the instruction set are the **`mfhc1/mthc1`** instructions described in Table 9-1 ''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 24KE core is described in Chapter 8 ''Floating point unit''.

Appendix A: References

MIPS Technologies manuals

[24KUSER]:
"MIPS32 24K Processor Core Family Software User's Manual", MIPS Technologies document MD00343.

[CorExtend]:
"How To Use CorExtend™ User-Defined Instructions", MIPS Technologies document MD00333.

[CorExtend-24K]:
"MIPS32 24K Pro Series™ CorExtend Instructions Integrator's Guide", MIPS Technologies document MD00348.

[EJTAG]:
"EJTAG Specification", MIPS Technologies document MD00047.

[MIPS32]:
the MIPS32 architecture definitions, 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.

[MIPS16e]:
"The MIPS16e™ Application-Specific Extension to the MIPS32 Architecture", MIPS Technologies document MD00074.

[MIPS64]:
the MIPS64 architecture definition. Although the 24KE core is a MIPS32 CPU, its floating point unit is described in these documents:

[MIPS64V1]:
"Introduction to the MIPS64 Architecture", MIPS Technologies document MD00081.

[MIPS64V2]:
"The MIPS64 Architecture Instruction set", MIPS Technologies document MD00085.

[MIPS64V3]:
"The MIPS64 Privileged Resource Architecture", MIPS Technologies document MD00089.

[MIPSDSP]:
"The MIPS DSP Application-Specific Extension to the MIPS32 Architecture", MIPS Technologies document MD00372.

[DSPWP]:
"Effective Programming of the 24KE and 34K Cores for DSP Code", MIPS Technologies white paper, document number MD00475.

[PDTRACEUSAGE]:
"PDtrace and TCB Usage Guidelines", MIPS Technologies document MD00365.

[PDTRACETCB]:
"The PDtrace™ Interface and Trace Control Block Specification", MIPS Technologies document MD00439.

Books about MIPS® programming

[SEEMIPSRUN]:
"See MIPS Run", author Dominic Sweetman, Morgan Kaufmann ISBN 1-55860-410-3. A general and wide-ranging programmers introduction to the MIPS architecture.

[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

These files are available as part of the free-for-download "SDELite" subset available from MIPS Technologies' website. You'll find them under `.../sde/include/mips/`.

[m32c0.h]:
the C definitions referred to in this manual for the names and fields of standard MIPS32 CP0 registers.

[mt.h]:
the C definitions for CP0 registers and other programmable resources of the MIPS MT.

Appendix B: CP0 registers by name and number

Register No./Set	*Register Name*	*Function*	*See ref/ section*
0.0	**Index**	Index into the TLB array	[MIPS32]
1.0	**Random**	Randomly generated index into the TLB array	
2.0	**EntryLo0**	Low-order portion of the TLB entry for even-numbered virtual pages	Fields in Figure 3-5, p. 25, more in [MIPS32].
3.0	**EntryLo1**	Low-order portion of the TLB entry for odd-numbered virtual pages	
4.0	**Context**	Pointer to page table entry in memory	[MIPS32]
5.0	**PageMask**	Control for variable page size in TLB entries	Section 3.5.1, p. 25
6.0	**Wired**	Controls the number of fixed ("wired") TLB entries	[MIPS32]
7.0	**HWREna**	Select which hardware registers are readable using the **rdhwr** instruction in user mode.	Section 9.1.2, p. 75
8.0	**BadVAddr**	Reports the address for the most recent TLB-related exception	
9.0	**Count**	Free-running counter at half pipeline speed.	Section 4.1.1, p. 27
10.0	**EntryHi**	High-order portion of the TLB entry	Figure 3-5, p. 25
11.0	**Compare**	Timer interrupt control	
12.0	**Status**	Processor status and control	Section 4.1.2, p. 27
12.1	**IntCtl†**	Setup for interrupt vector and interrupt priority features.	Section 9.5, p. 80
12.2 12.3	**SRSCtl†** **SRSMap†**	Shadow register set selectors In VI (vectored interrupt) mode, determines which shadow set is used for each interrupt source.	Section 9.6, p. 83
13.0	**Cause**	Cause of last general exception	Figure 9-3, p. 81
14.0	**EPC**	Restart address from last exception	[MIPS32]
15.0	**PRId**	Processor identification and revision	Figure 2-5, p. 17
15.1	**EBase**	Exception entry point base address and CPU ID (among multiple processors)	Figure 9-1, p. 78
16.0	**Config**	Configuration register	Figure 2-1, p. 13
16.1 16.2 16.3	**Config1** **Config2** **Config3†**	Standard configuration registers	Figure 2-2, p. 15 Figure 2-3, p. 16
16.7	**Config7**	24KE family-specific configuration	Figure 2-4, p. 16
18.0 18.1	**WatchLo0** **WatchLo1**	I-Watchpoint address	Section 5.3, p. 46
18.2 18.3	**WatchLo2** **WatchLo3**	D-Watchpoint address	
19.0 19.1	**WatchHi0** **WatchHi1**	I-Watchpoint control	
19.2 19.3	**WatchHi2** **WatchHi3**	D-Watchpoint control	
23.0	**Debug**	EJTAG Debug register	Figure 5-2, p. 34
23.1 23.2	**TraceControl** **TraceControl2**	PDtrace logic s/w setup	Figure 5-10, p. 43
23.3	**UserTraceData**	Write s/w-generated data for trace probe	Section 5.2, p. 44
23.4-5	**TraceIBPC** **TraceDBPC**	Control EJTAG breaks used as trace triggers (separate I- and D-side controls)	Figure 5-11, p. 44
24.0	**DEPC**	Restart address from last EJTAG debug exception	Section 5.1.5, p. 34
25.0 25.1 25.2 25.3	**PerfCtl0** **PerfCnt0** **PerfCtl1** **PerfCnt1**	Performance counter 0 control Performance counter 0 Performance counter 1 control Performance counter 1	Section 5.4, p. 46
26.0	**ErrCtl**	Software test enable for cache RAM arrays Controls access to way-select bits, parity, precode, BHT?	Section 3.2.3, p. 19
27.0	**CacheErr**	Cache parity error control and status	

† New with Release 2 of the MIPS32 specification.

Register No./Set	*Register Name*	*Function*	*See ref/ section*
28.0 28.2 28.4	**TagLo0** **TagLo1** **TagLo2**	Cache tag read/write interface for I-, D- and L2 cache respectively	Section 3.3.5, p. 22
28.1 28.3 28.5 29.1	**DataLo0** **DataLo1** **DataLo2** **DataHi0**	Cache low-order data read/write interface for I-, D- and L2 cache respectively... ... and high-order data for the I-cache, which is only accessible in 64-bit units.	
30.0	**ErrorEPC**	Program counter at last error	[MIPS32]
31.0	**DESAVE**	EJTAG debug exception save register	[EJTAG]

CP0 registers by name

Register Name	*Number*	*Register Name*	*Number*	*Register Name*	*Number*	*Register Name*	*Number*
BadVAddr	8.0	**DataHi0**	29.1	**IntCtl**	12.1	**TraceControl2**	23.2
CacheErr	27.0	**DataLo0**	28.1	**PRId**	15.0	**TraceControl**	23.1
Cause	13.0	**DataLo1**	28.3	**PageMask**	5.0	**UserTraceData**	23.3
Compare	11.0	**DataLo2**	28.5	**PerfCnt**	25.1	**WatchHi0**	19.0
Config1	16.1	**Debug**	23.0	**PerfCtl**	25.0	**WatchHi1**	19.1
Config2	16.2	**EBase**	15.1	**Random**	1.0	**WatchHi2**	19.2
Config3	16.3	**EPC**	14.0	**SRSCtl**	12.2	**WatchHi3**	19.3
Config7	16.7	**EntryHi**	10.0	**SRSMap**	12.3	**WatchLo0**	18.0
Config	16.0	**EntryLo0**	2.0	**Status**	12.0	**WatchLo1**	18.1
Context	4.0	**EntryLo1**	3.0	**TagLo0**	28.0	**WatchLo2**	18.2
Count	9.0	**ErrCtl**	26.0	**TagLo1**	28.2	**WatchLo3**	18.3
DEPC	24.0	**ErrorEPC**	30.0	**TagLo2**	28.4	**Wired**	6.0
DESAVE	31.0	**Index**	0.0	**TraceBPC**	23.4		

By Function

Basic modes	**Status**	12.0
Exception control	**Cause**	13.0
	EPC	14.0
	BadVAddr	8.0
Timer	**Count**	9.0
	Compare	11.0
Configuration	**PRId**	15.0
	Config	16.0
	Config1	16.1
	Config2	16.2
	Config3	16.3
	Config7	16.7
TLB maintenance (only if TLB)	**Context**	4.0
	EntryHi	10.0
	EntryLo0	2.0
	EntryLo1	3.0
	PageMask	5.0
	Index	0.0
	Random	1.0
	Wired	6.0
Cache	**TagLo0**	28.0
	TagLo1	28.2
	TagLo2	28.4
	DataLo0	28.1
	DataLo1	28.3
	DataLo2	28.5
	DataHi0	29.1
EJTAG debug/PDtrace logic	**Debug**	23.0
	DEPC	24.0
	DESAVE	31.0
	TraceControl	23.1
	TraceControl2	23.2
	UserTraceData	23.3
	TraceBPC	23.4
debug/ analysis	**PerfCtl0-1**	25.0/2
	PerfCnt0-1	25.1/3
	WatchHi0-3	19.0-3
	WatchLo0-3	18.0-3
regulate user-mode access to hardware registers	**HWREna**	7.0
interrupt setup	**IntCtl**	12.1
	EBase	15.1
shadow register setup	**SRSCtl**	12.2
	SRSMap	12.3
Parity/ECC control	**CacheErr**	27.0
	ErrCtl	26.0
	ErrorEPC	30.0

Appendix C: User instructions added for the MIPS32® Architecture

If you're familiar with CPUs that support legacy MIPS architectures, but not the MIPS32 architecture, this table summarizes the main changes to the instruction set. It ignores the widespread differences in the privileged-only CP0 register set.

The MIPS32 instructions are typically not used in most MIPS Linux OS versions, for example. But - if that's you - you'll surely need to read [MIPS32].

Table C.1 MIPS32® instructions which are not in all MIPS® ISAs

Instruction(s)	*Description*
`clo` `clz`	Count leading ones or zeroes.
`COP2`	Assembler cover for co-processor 2 instructions - the encodings always existed for anyone who'd use it. CP2 in MIPS Technologies cores is for user-defined extensions.

`madd`
`maddu`
`msub`
`msubu`
	Integer multiply/accumulate instructions (and negative versions), very valuable in "DSP" applications. Results accumulate in the `hi` and `lo` "registers" of the multiply/divide unit.

`movf`
`movn`
`movt`
`movz`
	Conditional move instructions; useful to avoid branches in expression evaluation, and particularly relevant to longer pipelines where branches are relatively expensive.

`mul`	3-operand 32-bit multiply; early MIPS CPUs always generated a result in the multiply/divide unit and required an explicit **`mflo`** to get it out.
`pref` `prefx`	Data prefetch hints, which typically work by bringing a line into the data cache. Where a program can identify data which is likely to cache miss (such as the first access to a large array) ingeniously placed prefetches can improve performance substantially. **`prefx`** uses the two-registers-added addressing format otherwise only seen in floating point instructions such as **`lwxc1`**, and is most likely to be used when prefetching floating point values. A field in the instruction permits a program to specify what it intends to do with the data, to permit further optimizations. You can find out about **`pref`** in the 24KE core by turning to Section 6.3 "Prefetching data".
`wait`	Put the CPU into a power-saving state, from which it will emerge when something happens (like an interrupt). There's a CPU-dependent hint field.

Appendix D: Revision History

Revision	Date	Description
1.00	4th July 2005	For ‘‘GA’’ release of the MIPS 24KE core family. Matches revision 1.00 of the MIPS DSP ASE specification.
1.10	21st December 2005	Update for maintenance release of the MIPS 24KE core family. Added 8K cache option and improved description of scratchpad RAM. Change bars are against v1.00.