MIPS® MT Principles of Operation

Document Number: MD00452
Revision 1.00
September 28, 2005

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

Template: S1.14, Built with tags: 2B

1 Background

The MIPS Multithreading (MIPS MT) ASE is defined in a rather complex specification document, the details of which can sometimes obscure the general principles of the architectural extension. This document is intended to serve as an introduction to the MIPS MT specification.

2 What is Multithreading?

As processor operating frequency increases, it becomes increasingly difficult to hide latencies inherent in the operation of a computer system. A high-end synthesizable core taking 25 cache misses per thousand instructions (a plausible value for “multimedia” code) could be stalled more than 50% of the time if it has to wait 50 cycles for a cache fill.

More generally, individual computer instructions have specific semantics, such that different classes of instructions require different resources to perform the desired operation. Integer loads don’t exploit the logic or registers of a floating-point unit, any more than register shifts require the resources of a load/store unit. No single instruction consumes all of a computer’s resources, and the proportion of the total system resources that is used by the average instruction diminishes as one adds more pipeline stages and parallel functional units to high-performance designs.

Multithreading arises in large measure from the notion that, if a single sequential program is fundamentally unable to make fully efficient use of a processor’s resources, the processor should be able to share some of those resources among multiple concurrent *threads* of program execution. The result does not necessarily make any particular program execute more quickly - indeed, some multithreading schemes actually degrade the performance of a single thread of program execution - but it allows a collection of concurrent instruction streams to run in less time and/or on a smaller number of processors.

2.1 Multithreading can provide benefits beyond improved multitasking throughput, however. Binding program threads to critical events can reduce event response time, and thread-level parallelism can, in principle, be exploited within a single application program to improve absolute performance.**Varieties of Multithreading**

There are a number of implementation models for multithreading that have been proposed, some of which have been implemented commercially.

Interleaved Multithreading is a TDM-style approach which switches from one thread to another on each instruction issued. Interleaved multithreading assures some degree of “fairness” in scheduling threads, but implementations which do static allocation of issue slots to threads generally limit the performance of a single program thread. *Dynamic* interleaving ameliorates this problem, but is more complex to implement.

The diagram below shows how instructions from program threads “A” and “B” might be issued. In the classical scalar RISC case, we see 4 consecutive instructions from program A being issued, with one wasted cycle due to a pipeline stall. A statically interleaved scheme which alternates between two threads might only be able to issue three instructions in the same amount of time, if A is the only thread available to run, as shown in the left-hand column, but if A and B are both runnable, as shown in the right hand column, the alternance between the two fills the pipeline and hides the stall in A. A dynamic interleaving scheme allows a single thread to run without degradation relative to the scalar RISC case, while still achieving better efficiency if multiple threads can be run.

Time	Scalar RISC	Statically Interleaved		Dynamically Interleaved	
		One Thread	Two Threads	One Thread	Two Threads
	A	A	A	A	A
	A	-	B	A	B
	A	A	A	A	A
	(stall)	-	B	(stall)	B
	A	A	A	A	A

Blocked Multithreading issues consecutive instructions from a single program thread until some designated blocking event, such as a cache miss, causes that thread to be suspended and another thread activated.

The diagram below shows how instructions from program threads “A” and “B” might be issued in a blocked multithreading system, relative to a scalar RISC. The scalar RISC stalls for as long as it takes to perform the cache refill, shown here as a wildly optimistic three cycles. A blocked multithreading design might behave identically to the scalar RISC if there is no other thread to run, but given two threads, the blocked multithreading processor switches from thread A to thread B as soon as thread A encounters the major stall. Note that the thread switch may not be instantaneous, and that while thread A is runnable on the last cycle shown, thread B retains the processor, since it has not yet been blocked.

Time	Scalar RISC		Blocked Multithreading		
			One Thread	Two Threads	
	A	Cache miss	A	A	Cache Miss
	(stall)		(stall)	(stall)	Thread switch delay
	(stall)		(stall)	B	
	(stall)		(Stall)	B	
	A		A	B	A ready, but B not blocked

Because blocked multithreading changes threads less frequently, its implementation can be simplified. On the other hand, it is less “fair” in scheduling threads. A single thread can monopolize the processor for a long time if it is lucky enough to find all of its data in the cache.

Hybrid scheduling schemes combining elements of blocked and interleaved multithreading have also been built and studied.

Simultaneous Multithreading is a scheme implemented on superscalar processors wherein instructions from different threads can be issued concurrently.

The diagram below shows a superscalar RISC, issuing up to two instructions per cycle, and a simultaneously multithreaded superscalar pipeline, issuing up to two instructions per cycle from either of the two threads. Those cycles where dependencies or stalls prevented full utilization of the processor by a single program thread are filled by issuing instructions for another thread.

Simultaneous multithreading is thus a very powerful technique for recovering lost efficiency in superscalar pipelines. It is also the most complex multithreading system to implement. More than one thread may be active at a given pipeline stage on a given cycle, complicating the implementation of memory access protection, etc.

Superscalar RISC

Time

A	A
A	-
(stall)	-
A	A
A	-

Simultaneous Multithreading, 2 Threads

A	A
A	B
B	B
A	A
A	B

2.2 Multithreading versus Multicore/multiprocessors

Multithreading and multiprocessing are closely related. Indeed, one could argue that the difference is one of degree: Whereas multiprocessors share only memory and/or connectivity, multithreaded processors share those, but also share instruction fetch and issue logic, and potentially other processor resources. In a single multithreaded processor, the various threads compete for issue slots and other resources, which limits parallelism. Some “multithreaded” programming and architectural models assume that new threads are assigned to distinct processors, to execute fully in parallel.

In very-high-end processors like the Intel P4, the throughput improvement from the limited parallelism provided by a multithreaded processor seems to be quite good relative to the incremental silicon cost, figures like 65% more throughput in return for 5% more silicon have been claimed. It must be understood, however, that the silicon cost of multithreading can be much higher as a percentage of total area in a small embedded core, relative to a Pentium 4-class processor.

In the light of all this, the MIPS MT ASE strives to provide a framework both for the management of parallel threads on the same CPU and for the management of parallel threads across multiple cores, and indeed for the migration of threads from one multithreaded processor to another.

3 Philosophical Principles

The MIPS MT multithreading ASE follows a set of philosophical principles which provide that it be

1. *Scalable* - The ASE should be implementable on simple, small, and high-frequency cores but still enable high efficiency of utilization on large, complex designs.
2. *Migratable* - The ASE should allow threads to migrate from processor to processor to balance load, and be amenable to multicore/multithreaded hybrid processor implementations.
3. *Scheduling Agnostic* - The ASE should be independent of the thread scheduling policies and mechanisms employed, and lend itself to simultaneous, interleaved, or blocked multithreading.
4. *Virtualizable* - The physical resources which support multithreading should be invisible or abstracted to the user-mode code, such that software which consumes more resources than exist on a particular implementation can nevertheless execute correctly, given appropriate OS support.
5. *Run-time Efficient* - Basic operations of thread creation and destruction, and of inter-thread communication and synchronization, should be realizable in a minimal number of clock cycles, without OS intervention in the most probable cases.

Where these principles conflict with one another, reasonable compromises are made.

4 MIPS MT ASE Definitions

The MIPS MT ASE is an **application-specific extension** of the MIPS32/MIPS64 instruction set and privileged resource architecture, meaning that it is a true architectural superset.

A **thread of execution**, or **thread,** is a sequential MIPS32/MIPS64 ISA instruction stream. A conventional MIPS processor runs a single thread at a time.

A **thread context**, or **TC**, is the hardware state necessary to support a thread of execution. This includes a set of general purpose registers (GPRs), a program counter (PC), and some multiplier and coprocessor state.

A **multithreaded processor** implements more than one TC, and can have more than one thread active at a time.

A **virtual processing element**, or **VPE**, is an instantiation of the full MIPS32/MIPS64 ISA and privileged resource architecture (PRA), sufficient to run a per-processor OS image. A VPE can be thought of as an “exception domain”, as exception state and priority apply globally within a VPE, and only one exception can be dispatched at a time on a VPE. A conventional MIPS core embodies a single VPE.

A **virtual multiprocessor**, or **VMP**, is a collection of interconnected VPEs. A VMP may be a single multithreaded MIPS processor core which implements multiple VPEs, and allows them to execute concurrently. In principle, a VMP may also be composed of multiple RISC cores, each of which may or may not be multithreaded.

4.1 Threads versus VPEs

Why the distinction between threads and VPEs? Because there are two ways for software to approach multithreading, one which is easy, but relatively expensive in silicon support and limited in the leverage provided to applications, and another which is more difficult to program, but which provides leverage for finer degrees of parallelism at a lower cost in silicon.

VPE Parallelism is equivalent to symmetric multiprocessor (SMP) parallelism. This means that operating systems which know how to deal with SMP system configurations can easily be adapted to run multi-VPE cores, and that programs already written using SMP multithreading or multi-tasking can exploit VPE parallelism.

Thread Parallelism in the context of the proposed ASE refers to fine-grained, explicitly controlled thread parallelism. This requires new OS/library/compiler support, but takes full advantage of low thread creation and destruction overhead to exploit parallelism at a granularity that would otherwise be impractical. The hardware support requirement for a TC is less than that of a VPE, so more TCs can be instantiated per unit of chip area.

4.2 Threads and Shadow Register Sets

Thread context storage on a MIPS processor can be used either to instantiate multiple parallel threads, or optionally to implement MIPS32 Release 2 shadow register sets (SRSs).

A TC is assigned to an SRS by writing the its number into the field of a new SRS configuration register (SRSConf0...4) which corresponds to the shadow set (1..15) by which the TC’s storage will be referenced.

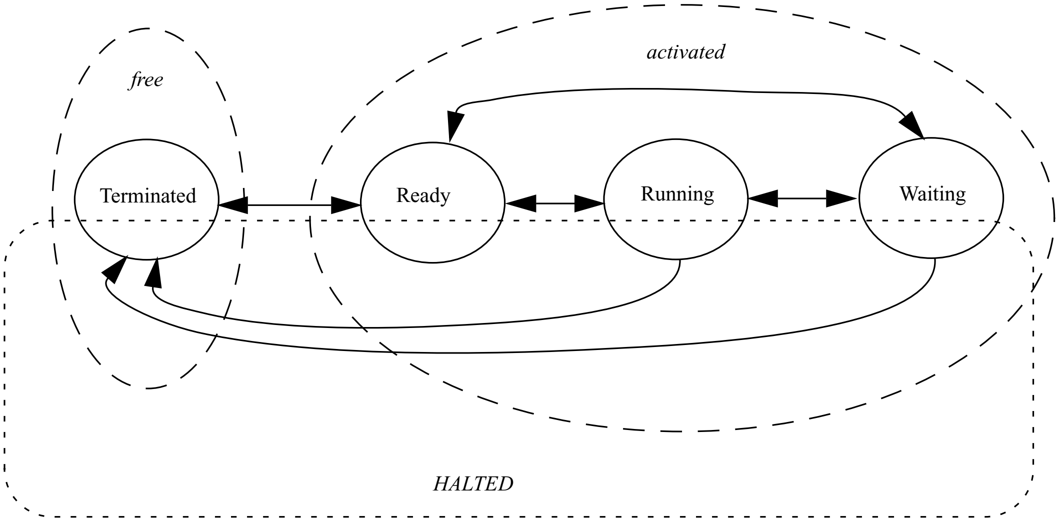
If an SRS is bound to an exception, it will be used by the handler for that exception, regardless of which TC is being used by the thread of execution causing the exception.

4.3 Implementation and Instantiation of Thread Contexts

The “name space” for TCs in MIPS MT is flat, with thread numbers ranging from 0 to 255. This does not mean that 256 TCs need to be implemented, nor that the full implemented compliment of architecturally visible threads needs to be instantiated as high speed, multi-ported register files. Designers may implement a hierarchy of thread storage, provided that the software semantics of the ASE are respected. A typical implementation would be a flat structure of 4-8 TCs.

4.4 Thread Context State Versus Thread Scheduling State

It is important to distinguish between the software-visible state of a TC as defined by the MIPS MT ASE, and the hardware state associated with the selection and scheduling of runnable threads. As seen by software, a MIPS MT TC may be in either a *free* or an *activated* allocation state, and independent of its allocation state, it may be *halted*, but an activated TC should not be confused with a “running” thread, though a running thread must have an activated context, nor should a halted TC be confused with a “waiting” thread. The following diagram shows the TC resource states superimposed on an implementation’s thread scheduling state machine.



5 The Multithreading ASE Instructions

The MIPS MT ASE extends both the instruction set and the privileged resources of MIPS32/MIPS64 architecture. The PRA extensions allow an operating system to manipulate the hardware resources associated with multiple VPEs and threads. The ISA extensions allow for efficient user-mode creation and destruction of threads so as to not require OS intervention in the typical cases.

5.1 Thread Creation

Threads can be directly created in user mode by using the new **FORK** instruction. The FORK instruction takes three operands. The first is the instruction address at which the new thread will begin execution, the second is an arbitrary register value to be passed to the new thread, typically a pointer to a block of thread-specific storage, and the third, an output parameter, is the register in the new thread's TC which will receive that second input operand value.

Design Considerations:

- **No Implicit Context Copy**. Some high-level multithreading software paradigms require that each new thread receive a copy of the full register set. Others do not. A MIPS32 user mode thread context consists of 31 GPRs (register 0 being always 0), the Hi/Lo or ACX/Hi/Lo accumulator, and some coprocessor state. Copying the full state in a single cycle would require an unrealistically wide interconnect between TCs. Copying sequentially, at a rate of 1-2 registers per cycle, would be complex and time consuming. The FORK instruction passes a single GPR value parameter to the newly spawned thread. If more than a single value is required for the computation, this value can be a pointer to a context block in memory that will contain register and other values needed by the thread computation. A FORK also implicitly propagates some privileged state, such as the contents of the ASID register, but the objective is to minimize the "payload" of a FORK operation, in part to minimize the hardware implementation cost, but also in anticipation of multicore "remote" FORKs, where the information would need to be transmitted between processing elements.
- **No Value Provided to Forking Thread**. Some high-level multithreading paradigms require that thread creation return a value to the "parent" thread performing the fork operation. This value is use as a handle or tag for future operations which may reference the thread. The proposed FORK instruction does not do this, for two reasons. First, it would require a register-file write beyond the write of the GPR value parameter to the register file of the new thread, which creates an undesirable constraint on the design of multithreaded register files. Second, it creates a name-space problem. In the course of its lifetime, the newly created thread may end up executing on different register sets of the same CPU, due to context switching, and it may even be migrated to some other processing element. Having hardware provide, as an output of the FORK instruction, a system-unique identifier that would follow each new thread, would be possible, but rather complex to impose on small, embedded cores. Where traceability is required, it can be accomplished using software-based memory interactions.
- **Absolute Virtual Thread Starting Address**. The FORK instruction takes as an input operand a register value which is taken to be the starting instruction fetch address for the new thread. Two alternative semantics were considered and rejected. It would have been possible to define FORK such that the new thread simply begins executing at the address following that of the FORK instruction. The two threads could be distinguished by data addressable through the parameter register value passed by the FORK. That would imply that the parent, as well as the child thread, perform the load, evaluation, and control transfer, which was rejected as wasteful. It would also have been possible to have the FORK instruction contain a "branch" offset in its encoding, rather than specify a register containing a full jump address. This would have made forks to thread code located quite close to the FORK instruction more efficient, but would have penalized all other FORKs by forcing them to perform a double control transfer.

Issuing a FORK instruction when there are no free, dynamically allocatable TCs available on the VPE causes a Thread "overflow" exception that can be used by system software to virtualize threads. A program can thus create and use a larger number of software threads than the available compliment of TCs, so long as the OS provides higher level scheduling to swap them in and out.

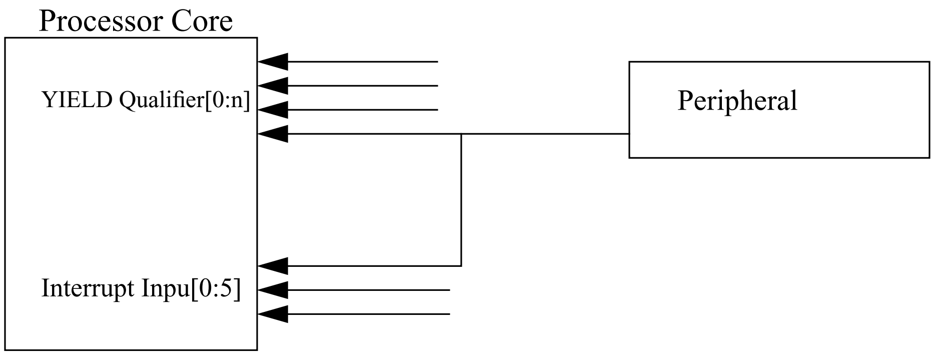
Issuing a FORK instruction when multi-TC scheduling is inhibited on the issuing VPE does not necessarily result in a failure or exception. So long as a TC can be successfully allocated, it is set up to run by the FORK operation, and will begin execution once TC scheduling is enabled.

5.2 Thread Suspension and Destruction

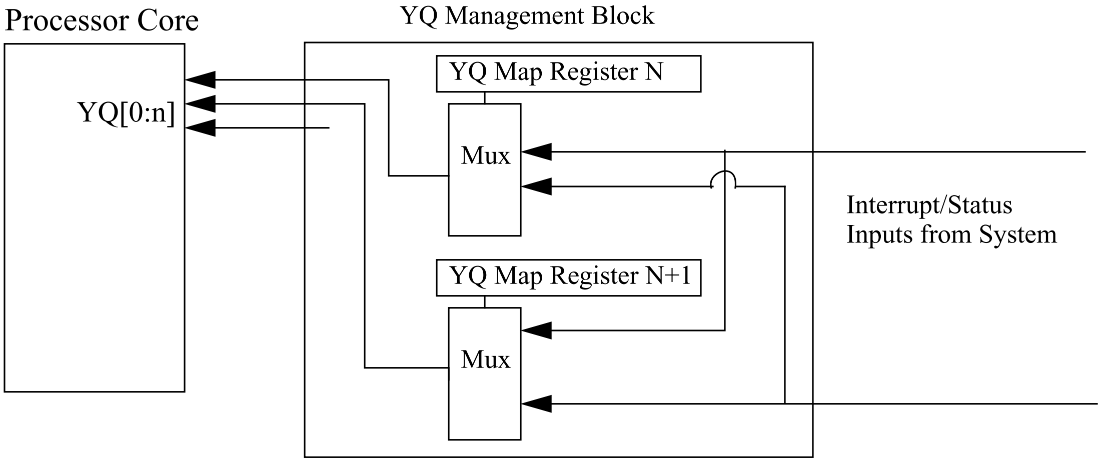
Thread execution is suspended by the **YIELD** instruction, which takes as an input operand a descriptor of the circumstances under which the issuing thread should be resumed. If the operand has a value of zero, there are no circumstances under which the thread will resume, and it is de-allocated so that the associated TC may be re-used.

Negative descriptor values are reserved by MIPS for architecturally defined rescheduling conditions. A value of -1 requests that the YIELDing thread be rescheduled without waiting for any specific condition, but allowing other threads to "play through", according to the implemented thread scheduling scheme. A value of -2 samples the YIELD Qualifier inputs (see below) without any rescheduling of the thread.

Positive descriptor values represent a vector of up to 31 independent "YIELD Qualifier" bits which are hardware inputs to the processor. In simple systems, these qualifier inputs may be directly connected to sources such as interrupt inputs, as shown below.



In more complex system configurations, incoming signals may be filtered and multiplexed by intermediary logic, configurable by software to map system signals to YIELD qualifiers as required by the application.



If interrupt request signals are brought into the YIELD qualifiers as described, even an interrupt condition that is masked by software so that it will not cause any exception vector to be executed can be used to enable the rescheduling of a service thread, for "zero latency" handling. On processors and VPEs using fine-grained multithreading, this has the additional advantage of not causing the processor/VPE to suspend multithreading due to entry by one thread into an exception handler.

Because positive-valued YIELD input parameters may describe combinations of possible events, any which may re-enable execution, the YIELD instruction produces a register value which contains implementation-specific information, such as which of several possible gating events cause the resumption of execution.

A Yield 0 which would terminate and deallocate the last dynamically allocatable TC on a VPE, such that none are left to execute, causes a Thread exception with an indication of the “underflow” condition.

The MIPS MT ASE provides mechanisms for an operating system to intercept and emulate YIELD operations. If a per-VPE enable is set, rescheduling YIELDs trap to the operating system with a designated exception code and subcode identifying a YIELD scheduler intercept. The operating system can evaluate the current YIELD qualifier input state, check it against the contents of the register specified by the trapping YIELD instruction, and make its own determination whether the YQ input values (or some synthetic value) should be placed in the YIELD’s destination register, and the TC restarted at the instruction following the YIELD, or whether the TC contents should be swapped to memory and replaced with the context of another thread of execution.

Each TC has a status bit that is set whenever an instruction is completed for that TC, outside of low-level exception handlers. This bit has multiple software uses, and it is further used by hardware to enable the YIELD scheduler intercept exception. An operating system which wishes to allow a TC to resume and remain blocked on a YIELD after handling a YIELD scheduler intercept exception can clear this “DT” (Dirty Thread) bit before restarting the TC on the YIELD, and that particular TC will remain blocked until the YIELD qualifiers are satisfied, or until some other OS intervention takes place. If the YIELD completes due to the qualifiers being satisfied, the DT bit will be set, and the next blocking YIELD issued by that thread will trap if the YIELD scheduler intercept exceptions are still enabled.

5.3 Other New Instructions

While FORK and YIELD are the two primitives on which user-mode multithreading is based,. the proposed multithreading ASE also includes some privileged instructions to help manage thread and VPE resources.

- **MTTR** is a privileged, “COP0” instruction which moves information from a register of the issuing thread to a register of another thread context on the same processor.
- **MFTR** is a privileged, “COP0” instruction which moves information from a register of another thread context on the same processor to a register of the issuing thread.
- **EMT** is a privileged, “COP0” instruction which atomically enables multithreaded issue on a VPE.
- **EVPE** is a privileged, “COP0” instruction which atomically enables multi-VPE issue on a VMP.
- **DMT** is a privileged, “COP0” instruction which atomically disables multithreaded issue on a VPE.
- **DVPE** is a privileged, “COP0” instruction which atomically disables multi-VPE issue on a VMP.

EMT, DMT, EVPE, and DVPE are in fact instances of the “MFMC0” instruction, much like the Release 2 EI and DI instructions.

6 New Privileged Resources

Some new privileged resources are required to manage the multithreading capabilities of a VPE. Three registers are defined to be per-processor, common to all VPEs on a core:

- A **MVPControl** register which contains control bits for managing multi-VPE processors.
- The **MVPConf0** and optional **MVPConf1** registers contain information about global multithreaded processor resources which can be configured at boot time and bound to different VPEs.

A total of 11 registers are defined to be per-VPE, common for all TCs within the VPE:

- A **VPEControl** register which contains information about the configuration of threads within a VPE.
- The **VPEConf0** and **VPEConf1** registers contain per-VPE information about the multithreading resources available to the VPE.
- A **YQMask** register which allows certain YIELD qualifier bits (see section 5.2) to be masked, so that an attempt to suspend execution pending that state will result in an exception.
- The optional **VPESchedule** register allows for the hardware scheduling algorithms of a processor to be manipulated to guarantee some “quality of service” to VPEs with hard real-time requirements.
- The optional **VPEScheFBack** register is the counterpart to the VPESchedule register, providing per-VPE feedback from the core scheduler logic to system software.
- An optional **VPEOpt** register which provides control/status information for optional features, such as run-time cache partitioning.
- Five optional **SRSConf0**-**SRSConf4** registers allow for run-time binding of TCs to Release 2 Shadow Register Sets.

Seven registers are defined to be per-TC.

- A **TCStatus** register which contains privileged resource information per-thread, such as the Kernel/User state of the thread, or whether it has access to a coprocessor.
- A **TCBind** register which defines a TC’s binding to a VPE.
- A **TCRestart** register which contains the restart fetch and execution address of a TC.
- A **TCHalt** register which allows a TC to be put into or taken out of a halted state with a single register write.
- A **TCContext** register which is simply a storage register implemented per-thread, which allows the OS to have instant access to a value, typically a memory pointer such as a kernel stack pointer, that is unique per-thread.
- The optional **TCSchedule** register allows for the hardware scheduling algorithms of a processor to be manipulated to guarantee some “quality of service” to threads with hard real-time requirements.
- The optional **TCScheFBack** registers is the counterpart to the ThreadSchedule register, providing per-TC feedback from the core scheduler logic to system software.

7 Threads and Exceptions

By definition, VPE parallelism introduces nothing new in the handling of exceptions for single-threaded VPEs within a multi-VPE processor. In the explicit, fine-grained model, however, multiple threads of execution with multiple hardware thread contexts share common system coprocessor resources. This has a number of implications for hardware and software.

Since there is only one Cause register to contain the reason for an exception, a single VPE cannot manage concurrent exceptions. When a synchronous exception is provoked by a thread, as in the case of a TLB miss or a floating-point exception, the MIPS32/MIPS64 architecture stipulates that the EXL or ERL bits of the Status register be set, which block interrupts and further general exceptions from being taken. In MIPS MT, the setting of EXL/ERL also prevents the scheduling of other threads until it is cleared by the exception handler.

Short exception handling sequences like TLB miss handlers can reasonably be coded as for a non-multithreaded MIPS32/MIPS64 processor, and re-enable multithreading implicitly with the clearing of EXL by the ERET instruction. More complex exception handling sequences, such as OS system calls, may explicitly re-enable the concurrent execution of non-privileged application threads by clearing EXL once the Cause information has been acquired and saved by the OS.

On a synchronous exception, the TC associated with the instruction stream causing the exception is the one which is associated with the exception: If the exception is not bound to a shadow register set, the associated TC is used to execute the exception handler, and if a shadow register set is used, the associated TC is used as the “previous shadow set”. Asynchronous exceptions, such as interrupts, can be associated with any available activated TC, with the restriction that TCs used by real-time service threads may be designated as exempt from use by interrupt service routines by setting a the IXMT per-thread control bit.

If all activated TCs are explicitly blocked via YIELD instructions or uncompleted loads/stores of gating storage locations (see section 11.1), asynchronous exceptions, including Debug exceptions, must be associated with such a blocked TC. The associated handlers will be executed using the previously blocked context, aborting the YIELD or load/store, and the VPE will resume execution on an ERET by re-fetching and re-executing the YIELD or load/store. An aborted gating storage load or store must leave the state of the storage location as it would have been had the instruction never been issued.

EJTAG Debug exceptions are special in several regards with respect to MIPS MT. Like other exceptions, they execute within the context of a specific VPE, but whereas the setting of EXL or ERL by a “normal” exception inhibits thread scheduling only within the affected VPE, Debug mode execution inhibits thread scheduling across *all* VPEs of a virtual multiprocessor core. And whereas other asynchronous exceptions, such as interrupts, require a TC that is activated and not halted (though it may have been blocked) to process the exception, an asynchronous Debug exception, such as that caused by the assertion of a DINT signal by an EJTAG probe, can be serviced by any TC bound to the targeted VPE, regardless of its halt or activation state. This makes it possible for EJTAG-based debuggers to recover from otherwise completely fatal OS errors, such as halting all TCs.

8 Allocating Instruction Bandwidth for Real-time Threads

Consider an embedded system in which two distinct kinds of processing are taking place, real-time decompression of audio or video data, and operation of a graphical user interface. Using late 20th century technology, this might be accomplished by using two different processors, a “real time” DSP processor to handle the multimedia data and an “interactive” core which runs a multitasking operating system with window management, etc. MIPS multithreading technology would allow these two functions to be performed on the same processor, using VPE multithreading. The processor or core would be a VMP, where each node would run a different OS and application set, one for media streaming, the other for the user interface. Using separate threads or VPEs solves the problem of co-existence of two different software paradigms, but it does not, on its own, guarantee the real-time performance requirements in the same way as a dedicated processor.

MIPS MT defines registers which allow scheduling parameters and feedback to be passed between software and hardware at both the TC and the VPE levels. The ThreadSchedule and VPESchedule registers allow for implementation-specific per-TC or per-VPE “hints” to be passed to the processor’s scheduling logic, and the ThreadScheFBack and VPEScheFBack registers allow for implementation-specific scheduler feedback to be likewise provided for each TC or VPE.

The effects of the VPESchedule and ThreadSchedule register are hierarchical. A thread with a hard real-time requirement for 3/4 of a processor’s total issue bandwidth, for example, could be assured of this in a dual-VPE VMP if the VPESchedule registers assigned 7/8 of the total issue slots to the real-time VPE, whose ThreadSchedule registers in turn assigned 7/8 of that issue bandwidth to the critical real-time thread.

9 Interaction with Coprocessors and CorExtend UDI Blocks

9.1 Coprocessors

From the principle of scalability, it would seem unreasonable to require that as many coprocessor register sets be implemented as there are scalar integer TCs. The multithreading scheme needs to accommodate third-party and customer-designed coprocessors which may have a single register set, as many register sets as the processor core has threads, or some number between the two.

The hardware mechanisms by which a coprocessor context is bound to a MIPS MT thread context are implementation-dependent. As with all MIPS32/MIPS64 processors, the operating system must in any case set the appropriate per-TC coprocessor usable (CU) bit before instructions targeting the coprocessor may be issued. The act of setting a CU bit may in itself suffice to perform the context binding, or more elaborate software setup may be required.

If only for diagnostic purposes, it must be possible for software to determine which coprocessor context is bound to a TC. The mechanisms for doing so are specific to a given coprocessor design, one obvious approach being to provide distinct per-coprocessor-context values for a specific coprocessor control register.

For the system coprocessor, CP0, all thread-specific system coprocessor information is carried in explicitly thread-specific registers, and all other CP0 registers are implicitly shared among all threads and protected by the serialization of exception entry, as described in section 7, and by explicit OS code as necessary.

9.2 CorExtend UDI Blocks

“Pure” CorExtend UDI blocks are unaffected by multithreading, since they operate only on GPRs. A multithreaded core can and must see to it that the GPRs accessed correspond to the thread whose instruction stream contained the UDI opcode. Any and all threads may take advantage of a UDI block, and the UDI interface can and should be extended to allow expected latencies to be signalled to the processor core, to facilitate thread scheduling.

UDI blocks which contain state require special attention. If only a single instance of UDI internal state is implemented, that state is necessarily shared between all threads executing on the associated processor. Unlike coprocessors, there is not necessarily a “UDI available” privileged resource that can be set or cleared per-thread, though implementations may provide this, and are encouraged to do so. Otherwise, it is the responsibility of application software to avoid contention for UDI-private resources.

It is also recommended that the UDI interface to the core be extended to allow an identifier of the issuing thread context to be provided to the UDI block on each UDI instruction issued, so that implementors can build multithreaded blocks if they choose.

10 32 versus 64-bit Implementations

Wherever necessary, MIPS MT specifies instructions to operate over the size of the implemented register, rather than over 32 or 64 bits. This allows the same binary code to “do the right thing” on both MIPS32 and MIPS64 cores and consumes a minimum of decode resources.

11 Gating Storage and Data-Driven Execution

Data-driven programming models map well to multithreaded architectures. For example, threads of execution can read data from memory-mapped I/O FIFOs, and be suspended for as long as it takes for the FIFO to fill, while other threads continue to execute. When the data is available, the load completes, and the incoming data can be processed directly in the load destination register without requiring any I/O interrupt service, polling, or software task scheduling.

However, the base MIPS32/MIPS64 architecture has no provision for restartably interrupting a memory operation once it has been processed by the MMU. It would thus be impossible for the TC of a blocked thread to be used by an exception handler, or for an operating system to swap out and re-assign such a TC. MIPS MT therefore introduces the concept of "Gating Storage", memories (or memory-like devices) which are distinguished as potentially requiring abort and restart of loads or stores. The abort/restart capability may require explicit support from the processor/memory interface protocols.

The MIPS MT ASE provides mechanisms for an operating system to intercept and emulate thread scheduling on blocking gating storage references. If a per-VPE enable is set, all blocking gating storage loads and stores abort and trap to the operating system with a designated exception code and subcode identifying a gating storage scheduler intercept. The operating system can evaluate the global VPE and system state, and make its own determination of whether the TC contents should be swapped to memory and replaced with the context of another thread of execution.

As in the case of the YIELD scheduler intercept exceptions, gating storage scheduler intercept exceptions are gated by the per-TC "DT" (Dirty Thread) bit, which is set whenever an instruction is completed for the TC. This allows an operating system to selectively allow a TC to resume and remain blocked on a gating storage reference after having taken a gating storage scheduler intercept exception, by clearing the DT bit before having the TC resume execution at the blocking load or store instruction.

It should be noted, however that, unlike YIELD instructions, loads and stores may occur in branch delay slots. Normal procedures for restarting after an exception on an instruction in a delay slot would restart on the branch, which could cause the DT bit to be set prior to the load or store. If it cannot be guaranteed that gating storage references will never be in branch delay slots, an operating system wishing to allow such a reference to be blocked despite the per-VPE gating storage scheduler intercept exception being enabled must create a "trampoline" that performs the gating storage reference, then jumps to the branch target address, and restart the TC to resume execution at the trampoline after clearing the DT bit.

11.1 InterThread Communications Storage

Inter-Thread Communication" (ITC) storage is a special case of gating storage, and can be thought of as a physical address subspace with special properties. Each 64-bit location or "cell" within ITC space appears at multiple consecutive addresses, or "views", distinguished by bits [6:3] of the load/store target address. Each view can have distinct semantics. The fundamental property of ITC storage is that loads and stores can be precisely blocked if the state and value of the cell do not meet the requirements associated with the view referenced by the load or store. The blocked loads and stores resume execution when the actions of other threads of execution, or possibly those of external devices, result in the completion requirements being satisfied. As gating storage references, blocked ITC loads and stores can be precisely aborted and restarted by systems software.

This has several motivations.

1. Issue bandwidth is a critical resource on multithreaded processors. Whereas spinning on a lock in a true multiprocessor system wastes only the issue bandwidth of the processor waiting on the resource, in a multithreaded processor, the act of polling the lock on the resource consumes issue bandwidth needed by the program thread holding the lock, and further delays the release of the resource. A thread blocked waiting on a value in ITC storage consumes no issue bandwidth until the value is produced or consumed.
2. Using hardware synchronization reduces the overhead of inter-thread control and data exchanges and makes finer grained parallel computations economical. A well-behaved algorithm running on an optimal implementation can pass values between threads at the cost of a single pipelined load or store cycle for each thread.
3. It allows a “push model” of multiprocessor/multithread data flow to be implemented in a near-optimal way.

For example, in some views, cells within ITC space may be “Empty” or “Full”. A load from a cell which is Empty causes the thread issuing the load to be suspended until the cell is written to by a store from another thread. A store to a cell which is Full causes the thread issuing the store to be suspended until a previous value has been consumed by a load.

Such ITC storage can define independent Empty and Full conditions, rather than a single Empty/Full bit, in order to allow for FIFO buffered ITC storage. In a classical Empty/Full memory configuration, Empty would simply be the negation of Full. A FIFO cannot be both Empty and Full, but it can be neither Empty nor Full if it contains some data, but could accept more.

While one view could be straightforward empty/full synchronization for producers and consumers, another view could implement classical “P/V” semaphores. Other views might implement atomic fetch-and-increment or fetch-and-decrement operations without blocking, etc.

Appendix A of the MIPS MT Specification provides a reference model for an ITC store which supports both Empty/Full and atomic P/V semaphores, with both blocking and “try” views.

12 Some Software Considerations

12.1 Virtual Multiprocessor

Mainstream operating systems technology understands symmetric multiprocessing (SMP) reasonably well. Several Microsoft operating systems support SMP platforms, as does Linux. “Multithreaded” applications exist which exploit the parallelism of such platforms, using “heavyweight” threads provided by the operating system. The VMP model of the proposed MIPS multithreading ASE is designed to provide maximum leverage to this technology. A multithreaded processor, configured as 2 single-threaded VPEs, is indistinguishable to applications software from a 2-way SMP multiprocessor. The operating system would have no need to use any of the new instructions or privileged resources defined by the ASE. Only in the case of a dynamically configurable VMP would logic need to be added to the boot code to set up the various VPEs.

Each MIPS MT TC has its own interrupt “exemption” bit and its own MMU address space identifier (ASID), which allows operating systems to be modified or written to use a “symmetric multi-TC” (SMTC) model, wherein each TC is treated as an independent “processor”. Because multiple TCs may share the privileged resources of a single VPE, an SMTC operating system requires additional logic and complexity to coordinate the use of the shared resources, relative to a standard MIPS32/MIPS64 OS, but the SMTC model allows SMP-like concurrency up to the limit of available TCs.

While the default configuration of multiple VPEs in a MIPS MT processor provides each VPE with an independently programmable MMU, such that legacy SMP memory management code will work correctly, it is possible for software to configure the processor to share MMU TLB resources across all VPEs. This requires a level of coordination between “CPUs” (really TCs or VPEs) that is not present in legacy SMP operating systems, but allows for an advanced SMP/SMTC operating system to achieve a more favorable TLB miss rate.

12.2 Master/Slave VPEs

One or more VPEs on a processor may power-up as “master” VPEs, indicated by the MVP field of the VPConf0 register. A master VPE can access the registers of other VPEs by using MTTR/MFTR instructions, and can, via the DVPE instruction, suspend all other VPEs in a processor.

This Master/Slave model allows a multi-tasking master “application processor” VPE running an operating system such as Linux to dispatch real-time processing tasks on another VPE on behalf of various applications. While this could be done using an SMP paradigm, handing work off from one OS to another, MIPS MT also allows this to be done more directly.

A master VPE can take control of another VPE of the same processor at any time. Once a DVPE instruction has been issued by the master VPE, the slave VPE’s CP0 privileged resource state can be set up as needed using MTTR instructions targeting TCs that are bound to the slave VPE, the necessary instructions and data can be set up in memory visible to the slave VPE, one or more TCs of the slave VPE can be initialized using MTTR instructions to set up their TCRestart addresses (and indeed their GPR register values, if appropriate), and the slave VPE can be dispatched to begin execution using the configured TCs by the master VPE executing an EVPE instruction.

12.3 Explicit Fine-Grained Multithreading

Finer-grained multithreading can exploit parallelism at levels that cannot be efficiently addressed by heavyweight, OS-level multithreading. The proposed MIPS multithreading ASE allows threads of execution to be created and destroyed very inexpensively by user-mode code. This requires that the applications or underlying libraries be explicitly built or coded to use the new instructions, of course, but it also requires OS support. No operating system in production use today has to deal with the situation where threads of execution, and thus contexts to be saved, can be created or destroyed without knowledge of the OS.

OS support for the fine-grained, FORK/YIELD parallelism of the proposed ASE would need to include:

- Context switch code which dynamically checks the number of threads to be saved and restored each time a user task is switched.
- Fault handling code for Thread exceptions, which occur when there is an underflow or overflow of the number of available physical TCs.
- Allocation and memory management code for ITC storage, if present, as “special” memory.

If threads at runtime without OS intervention are to be able to take nested exceptions, it is anticipated that the ThreadContext register value of each TC will be unique and persistent. The OS start-up code would assign context storage for each TC on a processor, and insert a pointer to it into that thread’s ThreadContext register prior to that thread’s being made available for FORK allocation.

12.4 Automatic Fine-Grained Multithreading

Automatic parallelization algorithms can be employed in compilers to generate multithreaded code. This technique is the ultimate means by which a single C or Fortran program can be accelerated in terms of execution “clock time”. The necessary compiler techniques exist in the research and high-performance computing communities. They would need to be adapted to MIPS, and used in conjunction with the OS support described above for explicitly fine-grained multithreading.

12.5 “Virtualization” of Threads and Hybrid Scheduling

If more software threads are active in a system than TCs are available in a MIPS MT VPE, it is necessary to impose a layer of software scheduling on top of the hardware thread scheduling policy of the processor. MIPS MT contains architectural hooks to support this “virtualization” of threads.

As noted in section 5.1, a FORK instruction when no dynamically allocatable TCs are free to accept the new instruction stream will cause a thread overflow exception, which allows an operating system to detect the case of more software than hardware threads in systems where user-mode thread creation is allowed. If software threads are created only by the OS, the OS will of course be able to track the available resources without need for an exception.

So long as the number of software threads does not exceed the TC resources available, it is of no consequence from the standpoint of system performance whether a TC remains blocked on a qualified YIELD or a gating storage access, but when TCs are saturated, it becomes necessary to multiplex the software threads across the available TCs. This can be achieved using simple scheduling algorithms that time-slice threads, regardless of whether or not they are making forward progress, but for high efficiency, it is highly desirable to use blockages as an opportunity to schedule other software threads. MIPS MT provides the option for a VPE to take an exception whenever a YIELD could cause a rescheduling or whenever a gating storage access blocks.

If blockages will generally be of a short duration, generating exceptions on each blockage may not be desirable, and it may be better to allow TCs to be blocked for some period of time before swapping out their contents. An operating system can do this by periodically sampling and clearing the “dirty” bit associated with each TC, which is set whenever the state of the TC is modified by instruction execution. If the dirty bit remains clear after a sample interval, it may be deduced that the TC has been blocked for the full interval.

If blocked threads are being swapped off of TCs because a of an unsatisfied YIELD qualifier or an unready gating storage location, it is of value to the operating system to know when that situation has changed, so that the thread can be swapped back onto a TC intelligently. While the MIPS MT architecture does not provide for this explicitly, it can be done straightforwardly with an appropriate system design. Because these gating events may be asynchronous to any instruction stream, they are most simply modeled as interrupts. YIELD qualifier inputs may simply be connected to maskable interrupts, and gating storage controllers can provide maskable interrupts that are asserted when a transaction has completed that might have unblocked a thread.

Context switching of one TC by another is possible using MFTR/MTTR instructions, but for loading/storing contexts in memory, MFTR/SW and LW/MTTR pairs are only half as efficient as a TC marshalling its own contents using sequences of loads and stores its own registers. That operation can be imposed by one TC on another by Halting the TC to be swapped, extracting its ThreadStatus and TCRestart values with MFTR instructions, setting its ThreadStatus TKSU bits to allow kernel mode execution and writing the address of the appropriate context switch code (which knows where to find the ThreadStatus and TCRestart values) into its TCRestart with MTTRs, then un-Halting the TC.

12.6 Software Security

If dynamic FORK/YIELD thread creation and resource allocation is in use simultaneously in different security domains, i.e. by multiple applications or by both an OS and an application, there can be a risk of information "leakage" in the form of register values inherited by an application. It is the responsibility of a secure operating system to manage this risk. MIPS MT provides one simple mechanism to facilitate this task, a "dirty" bit associated with each TC, which can be cleared by software and which is set whenever the context is modified. An OS can initialize all TCs to a known "clean" state, and clear all associated dirty bits, prior to scheduling a task. On a task switch, those TCs which are dirty must be "scrubbed" to the clean state before another task can be allowed to allocate and use them.

If a secure operating system wishes to make use of dynamic thread creation and allocation for privileged service threads, the associated TCs must be scrubbed before they are freed for potential use by applications.

MIPS MT provides no mechanisms which would allow a guarantee that two independent, untrusted tasks running concurrently on the same VPE and executing FORK and YIELD instructions, will not exchange TC storage, and thus register values. Programs which cannot "trust" one another should be run on distinct VPEs.

12.7 Manipulation of Dynamic Allocation Properties of TCs

Each TC has an associated DA bit which makes it available for dynamic allocation by FORK instructions. The interactions of FORK and YIELD with the set of DA bits makes possible several TC management algorithms. Interrupt-exempt real-time threads may have the DA bit of their associated TC cleared so that a YIELD 0 of the last dynamically allocated thread will cause an underflow Thread exception on the YIELDing thread without perturbing the real-time thread execution, and without leaving the processor in a state where no interrupt-capable TCs are active.

In response to an overflow Thread exception on a FORK, where no more DA TCs are available, the OS can, after having saved a copy of the previous values, clear the DA bits of all TCs, so that the next YIELD 0 will cause an underflow Thread exception which can be used by the OS to restore DA bits and schedule a replay of the failed FORK.

13 Run-time Configuration of Threads and VPEs

So long as each application of a multithreaded MIPS core will have its own layout and IC mask set, design engineers have the option to pre-determine how many threads need to be assigned to however many VPEs are required to fit the software model of the target system. In order to allow a single mask set and thus a single component to be used across a wide range of applications, however, the proposed MIPS multithreading ASE includes an option to provide VPE configuration by software. These mechanisms allow the same physical core to be configured as multiple or single VPEs, each with one or more threads. A single core design could be used as a SMP multi-VPE platform in the near term, but still be able to benefit from fine-grained multithreading techniques at a later date, with an upgrade to the boot code, OS, and, of course, the application code.

“Master” VPEs as described in section 12.2 can enable VPE reconfiguration by setting the VPC bit of the MTConfig register. While the VPC bit is set, VPE parameters can be changed by writing to some Config0-3 and VPEConf1 register fields that are otherwise preset or read-only. Clearing the VPC bit latches the new configuration.