An Overview of MIPS Multi-Threading

White Paper

Filename	:	Overview_of_MIPS_Multi_Threading.docx
Version	:	1.0.3
Issue Date	:	19 Dec 2016
Author	:	Imagination Technologies

Contents

1. Motivations for Multi-threading 3

2. Performance Gains from Multi-threading 4

3. Types of Multi-threading 4

3.1. Coarse-Grained MT 4
3.2. Fine-Grained MT 5
3.3. Simultaneous MT 6

4. MIPS Multi-threading 6

5. R6 Definition of MT: Virtual Processors 7

5.1. Nomenclature 7
5.2. Hardware Resources Replicated per Virtual Processor 7
5.3. Hardware Resources Shared Among Virtual Processors 7
5.4. Implementation-specific Choices 7
5.5. Detection of Virtual Processor-MT Feature 9
5.6. Enabling and Disabling Virtual Processors 9
5.7. Virtual Processor Numbering 9
5.8. Software and Hardware States 9

6. Virtual Processors and Symmetric Multi-Processing 10

6.1. Synchronization primitives 10

7. Performance optimizations 10

7.1. Data-Driven Scheduling of Threads 10
7.2. Inter-thread Communication Unit 10

Appendix A. MT ASE – for R2 - R5, not used for R6 11

1. Motivations for Multi-threading

As a computer program is executed, there are many events that can cause the CPU hardware resources not to be fully utilized every CPU cycle. Such events include:

- Data Cache Misses – the required data must be loaded from memory outside the CPU. The CPU has to wait for that data to arrive from the remote memory.
- Instruction Cache Misses – the next instruction of the program must be fetched from memory outside the CPU. Again, the CPU has to wait for the next instruction to arrive from the remote memory.
- Data dependency stalls – the next instruction cannot execute yet as one of its input operands hasn't been calculated yet.
- Functional Unit stalls – the next instruction cannot execute yet as the required hardware resource is currently busy.

When one portion of the program (known as a thread) is blocked for one of these events, the hardware resources could potentially be used for another thread of execution. By switching to a second thread when the first thread is blocked, the overall through-put of the system can be increased. The idea of speeding up the aggregate execution of all threads in the system is known as "*Throughput Computing*". This is in contrast to speeding up the execution of a single thread (or known as single-threaded execution).

If one replicates an entire CPU to execute a second thread, then the technique is known as multi-processing.

If one replicates only a portion of a CPU to execute a second thread, then the technique is known as multi-threading.

A simple graphical example is shown in following figures. The Multi-threaded implementation in Figure 2 does more aggregate work in the same number of cycles as the single-threaded CPU in Figure 1. Instead of having the execution pipeline being idle while waiting for the Memory data to arrive, the Multi-threaded CPU executes code for Thread2 during those same memory access cycles. The idle cycles in black are often known as pipeline "bubbles".

A. Figure 1 Single thread execution on a single CPU pipeline

B. Figure 2 Multi-threaded execution on a single CPU pipeline

Sharing hardware resources among multiple threads gives an obvious cost advantage to multi-threading as compared to full-blown multi-processing. Another potential benefit is that multiple threads could be working on the same data. By sharing the same data caches, multiple threads get better utilization of these caches and better synchronization of the shared data.

By minimizing how much hardware is replicated for executing a software thread, Multi-threading can boost overall system performance and through-put with relatively little additional hardware cost. Because there is relatively less additional hardware, the performance gain is achieved with less additional power consumption.

2. Performance Gains from Multi-threading

The performance boost from Multi-threading comes from filling up all of the CPU cycles with productive work that otherwise would be un-used due to stalls. Many applications have low number of instructions-executed-per-cycle when run in single-threaded mode and are good candidates for multi-threading.

Any application which can keep the CPU fully busy every cycle with a single thread is not a good candidate for multi-threading. Such applications are relatively rare.

Since the introduction of the first MT enabled MIPS CPUs, there have been multiple studies of the MT performance benefits. The studies show a range of performance gains from 15% to 226%.

Refer to these Application Notes on MT performance gains: The following Application Notes are available here: https://community.imgtec.com/developers/mips/resources/application-notes/

- MD00535: Increasing Application Throughput on the MIPS32® 34K® Core Family with Multithreading.
- MD00547: Multi-threading Applications on the MIPS32® 34K® Core
- MD00545: Multi-threading for Efficient Set Top Box SoC Architectures
- MD00828: Optimizing Performance, Power and Area in SoC Designs using MIPS® Multi-threaded Processors
- MIPS Creator CI-40 Multithreading Benchmarks

3. Types of Multi-threading

3.1. Coarse-Grained MT

The simplest type of multi-threading is known as Coarse-Grained Multi-Threading (MT). For this type, one thread runs until it is blocked by an event that creates a long latency stall (normally an all-cache miss). This long latency stall has to be identified and checked by the programmer and then the processor is programmatically switched to run another thread.

Conceptually, it is similar to cooperative multi-tasking used in Real-Time Operating Systems, where one task realizes it is blocked and then hands off the execution time to another task.

This type of multi-threading is also known as Blocked or Cooperative Multi-threading.

Here is a simple example:

CPU Cycle	Thread being executed	Operation
i	ThreadA	Instruction j from ThreadA
i+1	ThreadA	Instruction j+1 (load instruction which misses all caches)
i+2	ThreadA	Instruction j+2 (check for cache miss for instr J+1)
i+3	Thread Scheduler	Cache miss detected; Thread scheduler invoked; switch to Thread B
i+4	ThreadB	Instruction k from ThreadB

CPU Cycle	Thread being executed	Operation
i+5	ThreadB	Instruction k+1

Hardware support for this type of multi-threading is meant to allow quick switching between the threads. To achieve this goal, the additional hardware cost is to replicate the program visible state – such as the GPRs and the program counter for each thread. For example, to quickly switch between two threads, the hardware cost would be having two copies of the GPRs and the program counter.

For this type of multi-threading, only long latency stalls can cause thread switches, as an instruction in the program has to be added for each stall check. It would be too costly to add such instructions to check for very short stalls.

3.2. Fine-Grained MT

A more sophisticated type of multi-threading is known as Fine-Grained Multi-Threading. For this type, the CPU checks every cycle if the current thread is stalled or not. If stalled, a hardware scheduler will change execution to another thread that is ready to run. Since the hardware is checking every cycle for stalls, all stall types can be dealt with, even single cycle stalls.

Early implementations of this type of multi-threading caused a thread switch **every** CPU cycle. The motivation for switching every cycle was to reduce the possibility of stalling for a previous result from the same thread. This early type was known as barrel processing, in which staves of a barrel represented the pipeline stages of the CPU. It was also known as interleaved or pre-emptive or time-sliced multi-threading. It was conceptually similar to preemptive multi-tasking, used in operating systems, where the time slice that is given to each active thread is one CPU cycle.

Here is an example of a barrel processor executing 3 threads in round-robin fashion:

CPU Cycle	Thread being executed	Operation	Comment
i	ThreadA	Instruction j from ThreadA	Thread switch every cycle
i+1	ThreadB	Instruction k from ThreadB	Thread switch every cycle
i+2	ThreadC	Instruction l from ThreadC	Thread switch every cycle
i+3	ThreadA	Instruction j+1 from ThreadB	Thread switch every cycle
i+4	ThreadB	Instruction k+1 from ThreadB	Thread switch every cycle
i+5	ThreadC	Instruction l+1 from ThreadC	Thread switch every cycle

The additional hardware cost of fine-grained Multi-threading is to track the Thread ID of the instruction in each pipeline stage. In addition because there are multiple threads that are concurrently active, shared resources such as caches and TLBs might need to be increased in size to avoid thrashing between the different threads.

More modern implementations would only cause a thread switch when the currently running thread becomes blocked. For these more modern implementations, a thread can continue executing until it would produce a stall.

Here is an example of this more modern type of fine-grained multi-threading:

CPU Cycle	Thread being executed	Operation	Comment
i	ThreadA	Instruction j from ThreadA	
i+1	ThreadA	Instruction j+1 from ThreadA	
i+2	ThreadA	Instruction j+2 from ThreadA	Detect instr j+3 would stall
i+3	ThreadB	Instruction k from ThreadB	Detect instr k+1 would stall
i+4	ThreadC	Instruction l from ThreadC	

3.3. Simultaneous MT

The most sophisticated type of multi-threading applies to superscalar processors. Superscalar means that the processor can execute multiple instructions in each CPU cycle.

Simultaneous Multi-threading (SMT) means that each of these instructions which are issued together can either be from the same thread or each can be from different threads. The hardware thread scheduler will pick the most appropriate instruction to maximize the utilization of the execution pipelines.

Here is an example of SMT execution on a dual-issue CPU:

CPU Cycle	Issue Slot 1	Issue Slot 2	Comment
i	ThreadA, instr j	ThreadB, instr k	2 instrs from different threads
i+1	ThreadB, instr k+1	ThreadB, instr k+2	2 instrs from same thread
i+2	ThreadC, instr l	ThreadA, instr j+1	2 instrs from different threads

4. MIPS Multi-threading

The first multi-threaded processor from MIPS was the 34K, which was released in 2005. The 34K implemented fine-grained multi-threading (the more modern kind which doesn't have to blindly switch threads every cycle), with a hardware thread scheduler within the CPU which picks the most appropriate thread to run each CPU cycle. All subsequent multi-threaded processors from MIPS have also implemented fine-grained multi-threading including 1004K, interAptiv, I6400, I6500.

The I6400 is a super-scalar CPU and is the first MIPS CPU which also implemented SMT. The I6500 also implements SMT.

5. R6 Definition of MT: Virtual Processors

The R6 versions of MIPS32/64 architectures (released in 2014) introduced a simplified definition for MIPS Multi-threading. In this simplified definition, the entity executing a software thread is known as a Virtual Processor. The following sections describe Virtual Processors and how they are used.

5.1. Nomenclature

A fully complete CPU is known as a **Physical CPU**. This includes the instruction fetch, instruction dispatch, execution pipelines, memory-management-unit, cache hierarchy, load-store unit, etc.

An entity that can execute a software thread is known as a **Virtual Processor**. This is a subset of a complete CPU, at a minimum, the programmable state that is usable by the thread – user registers, the program counter, etc. The other parts of the CPU - the execution pipelines, caches are shared among multiple Virtual Processors. In a multi-threaded CPU, a Physical CPU can host multiple Virtual CPUs.

5.2. Hardware Resources Replicated per Virtual Processor

The following are hardware resources which are replicated for each Virtual Processor:

- User-mode state such as GPRs and FPRs
- Kernel-mode state for interrupt and exception processing
- Kernel-mode state for control & status of the CPU
- Kernel-mode state for managing the MMU
- Reset logic
- EJTAG debug logic

5.3. Hardware Resources Shared Among Virtual Processors

The following are hardware resources which are shared among all Virtual Processors within one Physical Processor:

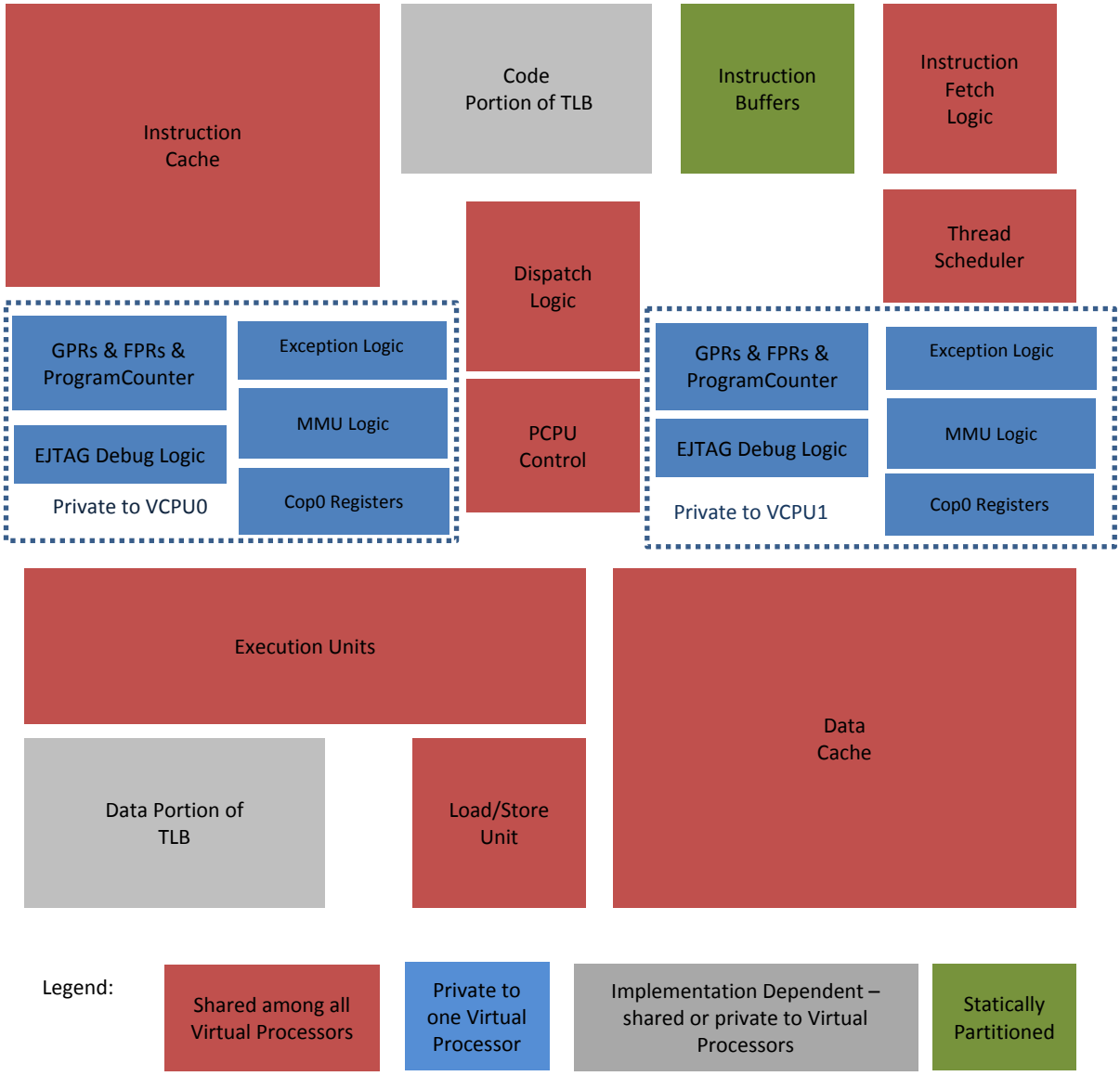
- The execution pipelines
- The cache hierarchy
- Instruction fetch and dispatch logic
- Load-store Unit

5.4. Implementation-specific Choices

Because the TLB in a high-end CPU might take a very large portion of the die-area, some implementations might want to share the TLB among all of the Virtual Processors, as opposed to replicating a TLB for each Virtual Processor. Implementations are allowed to make this choice.

Figure 3 is a high-level block diagram showing the units which are shared among all Virtual Processors; which are private to one VCPU.

Figure 3: Block diagram of Physical CPU with 2 Virtual Processors



5.5. Detection of Virtual Processor-MT Feature

Software can detect if the hardware supports the Virtual-Processor version of Multi-threading by reading the COP0.Config5.VP register field. Config5 is COP0 register 16, Select 5.

5.6. Enabling and Disabling Virtual Processors

Transitioning a **Physical CPU** from Single-threaded execution to multi-threaded execution is done by executing an ***EVP*** instruction.

Transitioning a **Physical CPU** from Multi-threaded execution to Single-threaded execution is done by executing a ***DVP*** instruction.

These instructions are privileged and can only be executed in the most privileged execution mode - kernel-mode (or Root-Kernel mode if MIPS Virtualization is supported). Otherwise, they cause an exception.

5.7. Virtual Processor Numbering

Each of the Virtual Processors has a unique system-wide identifier that is the combination of the Cluster Number, the (Physical) Core Number and the Virtual Processor Number. These values are held in the COP0.GlobalNumber register. GlobalNumber is COP0 register 3, Select 1. The GlobalNumber values do not change when (either Physical or Virtual) CPUs are power-downed or disabled.

Each of the active Virtual Processors also has an identifier in the COP0.EBase.VPNum field. EBase is COP0 register 15, Select 1. In some implementations, the Ebase.VPNum value can change when (Physical or Virtual) CPUs are power-downed or disabled. These value changes are to keep the CPU numbering to be contiguous even when a CPU transitions from active to non-active.

5.8. Software and Hardware States

Something to be aware of are the different states that are used in a MT-enabled CPU. There are Software states and Hardware states.

Software states are used to track the status of the thread. These might say whether the Thread is runnable/active or whether the thread is completed/merged with its parent. There might be additional states to say whether the thread is stalled/blocked waiting for an event to occur.

Hardware states are used to track the status of the processor which is meant to run the thread. These might say whether the processor is enabled or disabled (potentially for power-savings).

6. Virtual Processors and Symmetric Multi-Processing

Because Virtual Processors have most of the user-mode and kernel-mode software visible states replicated, software usually cannot tell the difference between a full multiprocessor and multiple Virtual Processors. That is, a Physical CPU with multiple Virtual Processors looks like a Symmetric Multiprocessor to most software.

6.1. Synchronization primitives

The synchronization primitives available to Virtual Processors are exactly the same as those used by Multiprocessors:

DI and ***EI*** instructions to disable/enable interrupts

LL and ***SC*** instructions to create synchronization primitives such as spin-locks and semaphores. Double width semaphores can be created by using the ***LLX*** and ***SCX*** instructions in sequence with the ***LL*** and ***SC*** instructions.

Wait and ***Pause*** instructions are available to put threads to sleep.

7. Performance optimizations

7.1. Data-Driven Scheduling of Threads

One powerful performance optimization that is available in a MIPS Multi-threaded system is data-driven scheduling of threads.

In this scheme, a thread is assigned to deal with some specific incoming data. When the data has not arrived yet, the thread is not active. The thread only becomes active when the data arrives.

The benefit of this thread scheduling scheme is avoiding the use of interrupts to notify the CPU of the arrival of the data. When interrupts are used, the CPU pipeline needs to be flushed and registers have to be context switched, all of which takes time.

To achieve this optimization, the arriving data has to send a signal to the waiting thread. The next section describes a hardware block that enables this signaling.

7.2. Inter-thread Communication Unit

The Inter-Thread Communication Unit is a specialized memory block. The ITCU is an optional HW block in MIPS MT-enabled processors.

In one mode of operation, this memory block acts like a set of FIFO queues. These FIFO queues are used as mail boxes for the threads.

The thread waiting for the data accesses the output register of the assigned FIFO queue.

If the FIFO is empty, a signal is sent to the thread to de-activate the thread. The Hardware thread scheduler then knows not to select this thread for issuing instructions in the next cycle.

When the data arrives in the assigned FIFO queue, a signal is sent to the HW thread scheduler to say the waiting thread is now active. The waiting thread thus becomes a valid candidate for its instructions to be executed in the next cycle.

This state transition of the waiting thread does not incur any cycle penalty of flushing the pipeline nor context switching the register files.

Appendix A. MT ASE – for R2 - R5, not used for R6

Prior to Revision 6 of the MIPS32/64 architecture, a more complicated definition of Multi-threading was used. This older definition is called the MIPS Multi-threading Application Specific Extension (also known as MT ASE or the MT Module as part of MIPS R5 base architecture). The MT ASE definition is a super-set of the Virtual Processor definition.

The 34K, 1004K and interAptiv products were designed according to the MT ASE. We will call these the “Legacy MT ASE products”.

The MT ASE breaks up the Virtual Processor into 2 pieces:

- The User-mode portion - comprising of the GPRs and FPRs, the program counter and a few COP0 register fields. This User-mode portion is called the “Thread Context” or “TC”.
- The kernel-mode portion - comprising of the exception logic, the MMU logic, the COP0 registers, the TLB and the EJTAG debug block. This kernel-mode portion is called the “Virtual Processing Element” or “VPE”.

If one TC is assigned to one VPE, then a Virtual Processor is created. This is exactly the same as the R6 Virtual Processor definition.

But in the MT ASE it is also allowed to assign multiple TC’s to one VPE. Such an entity looks like a CPU with multiple register files, with each register file assigned to a different thread.

The Legacy MT ASE products allowed both fine-grained and coarse-grain context switching of the threads. The MT ASE provided the ***FORK*** and ***JOIN*** instructions for the coarse-grain/cooperative thread switching.

This type of cooperative context switching requires a custom software execution environment that is different from the more familiar symmetric multi-processing environments.

Software can detect if the CPU implements the MT ASE by checking COP0.Config3.MT register field. If Muilti-Threading is supported by the MIPS CPU, one of COP0.Config3.MT or COP0.Config5.VP bits will be set, but not both bits.

Each Virtual Processor is enabled and disabled by the ***EVPE*** and ***DVPE*** instructions. Each Virtual Processor can execute with multiple TCs, this ability is enabled/disabled by the ***EMT*** and ***DMT*** instructions.

The MT ASE includes many COP0 registers for lower-level control of TC and VPE behaviors.