In this section I'll cover instructions added to the instruction set for the MT ASE

The MIPS® MT ASE: New Instructions

- FORK allocates a TC and sets it running
- YIELD causes rescheduling/reallocation
- MFTR moves values from a targeted TC to a GPR
- MTTR moves values to a targeted TC from a GPR
- DMT disables multithreading on a VPE
- EMT enables multithreading on a VPE
- DVPE disables multithreading across all VPEs on a processor
- EVPE enables multithreading across all VPEs on a processor

MIPS

2

There are 8 instructions that have been added to the instruction set for MT. I'll go into each in detail.

MT FORK Instruction

- **FORK causes a free and dynamically allocatable thread context, to be allocated and associated with a new instruction stream on the issuing VPE.**

  fork rd, rs, rt

  - *rs* points to the instruction where the new thread is to start execution.
  - The new thread's *rd* register gets the value from the existing thread's *rt* (way of passing information to the new thread).

MIPS

3

The fork instruction fires up a thread on a free TC.

+ rs is a pointer to the instruction where the new thread will start executing.

+ RT is the number of the current threads general purpose register who's value will be copied to the new threads

+ general purpose given in RD. This way you can pass a value the new thread by initializing one of its GPRs with the value of one of the current thread's GPRs.

MT FORK Instruction

- New thread inherits the execution mode, *TCStatus[TKSU] and* the address space *TCStatus[TASID].*
- To be successful:
  - There must be non-Active thread, (*TCStatus[A]* = 0)
  - The free thread must be Dynamically Allocatable, *TCStatus[DA]=1*
- Failure to find a free and allocatable thread results in a Thread Overflow exception.

MIPS

Some vital per-TC state is copied from the parent TC. The kernel or user mode state , defined in

TCStatus[TKSU], and what address space identifier defined in TCStatus[TASID]

+fork will only select a TC which is both "free" identified by the TCStatus register A field being zero

+ and the TC is specifically marked as usable by fork identified by the TCStatus register DA field being set.

+ If there are no TCs to allocate a Thread Overflow exception happens and it is left up to the OS to figure out what to do.

MT Yield Instruction

- **YIELD controls thread run status. Yield allows a thread to transition itself from a running state to a non-running state. The non-running state depends on the value of rs.**

Yield rd, rs

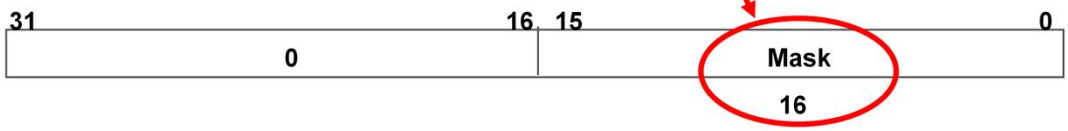
- rs = 0 Terminates the thread so the thread can be re-tasked when needed.

The Yield instruction allows a thread to control its execution. It can terminate itself, check for events to process or wait for events to happen.

+ A TC that issues a yield with rs as GPR $0 will terminate itself. This could result in a "Thread Underflow" exception, which occurs when you're about to reach a situation where all the are available for a fork are stopped, The exception is used to tell the OS that the system might stop forever, with no threads running the code so it can decide what to do.

MT Yield Instruction

- rs = -1 Threads way of communicating to the policy manager that it can be paused if need be according the preset policy.
- rs = -2 Threads way to get currently set YQ signals returned in rd.
- rs > 0 Thread will wait for up to 16 possible signals to be asserted. rs is used as a signal mask.



MIPS

6

When rs equals a -1 the thread maybe be stopped briefly while the yield indication is sent out to an external scheduling policy manager, if this feature is supported by the policy manager the policy manager may cause the thread to stop for a while, lower this threads priority or that other actions. This behavior is strictly the result of the implementation of the policy manager. None of the policy managers supplied by MIPS use this feature.

+ When rs equals a -2 the current value of all the Yield Qualifier signals are placed into the GPR number in RD. There is no scheduling effect, purely done for the value delivered to rd.

+ when rs greater than 0 the TC waits for one or more of the Yield Qualifier signals to be asserted and masked by a “1” bit in the rs GPR value. This feature can be used to tie a specific thread to a external event. Several threads can be setup to wait on different events depend on the mask given by rs.

MT Yield Instruction

- rs > 0 Continued
  - The accessibility of the signal bits is controlled by the YQMask register.
  - The signals are contained in the lower 16 bits of the register. The upper 16 bits of the register should be ignored as the bit values may float.
  - yield rd, rs – On return the rd register will contain all active signals that are accessible to the Yield instruction.
  - A yield instruction must not be in a branch delay slot.

MIPS

7

+ As I have previously covered in the MT control registers section the YQMask register enables particular Yield Qualifier pin signals so if the bit corresponding to the pin is not set a "invalid qualifier“ exception will be generated and the OS can then decide what to do with the thread.

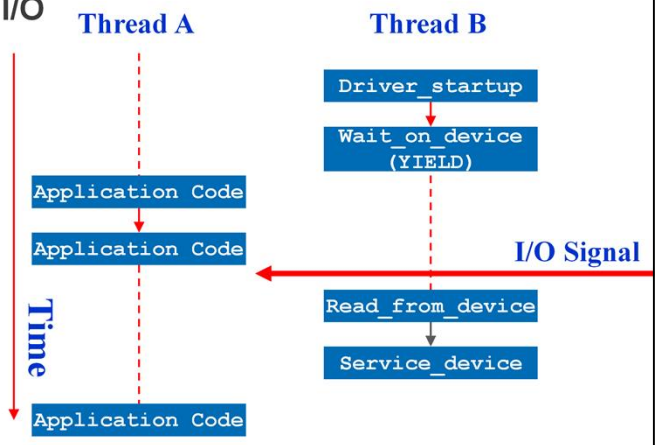
+ There are only 16 pin inputs so only the lower 16 bits of the YQMask register are used the rest may float.

+ After the Yield instruction completes and the TC starts executing again the GPR referenced by rd will contain the value of all active signals that are not masked by the YQMask register.

+ The yield instruction should never be put into a branch or jump delay slot.

Use rs > 0 To Eliminate Interrupt Response Time

- Direct Activation of pre-initialized I/O Service Threads
- Like Shadow Registers but with no vector fetch overhead
- "Zero" Interrupt Latency with Aggressive Thread Scheduling Policies
- *Use Where Interrupts are Very Frequent and service is Very Quick*
- Can Be Done in User Mode
- Works even if Interrupts are Disabled



MIPS

8

The Yield instruction with RS> 0 allow a thread to wait for an external event like a interrupt only in a much more direct fashion.

+ Since each thread has its own GRP set it is like using an interrupt with a Shadow Registers but with the advantage of no vector fetch overhead. There are no cycles or cache state lost to saving/restoring register context

+ this all adds up to no latency once a signal is asserted on a Qualifier pin.
No cycles lost to pipeline flushing and fetch redirection,

Because no exception is taken, vs. 10 cycles of pipeline bubble on an interrupt.

Post-YIELD instructions typically already in Instruction Buffer so thee are no additional fetch cycles

+ This method can by use to replace any interrupt but is particularly good to use in place of a interrupt where the interrupt is very frequent and the interrupt service routine is very quick.

+ the Yield instruction can be used in user mode so there is no need to switch to Kernel mode to execute it

+ interrupts do not have to be enabled

+ Say we have 2 threads in our system A and B

+ Thread B starts a I/O call and used the Yield instruction to Wait for the I/O to complete.

+ Thread A can then use all of the CPU cycles to execute instructions.

+ Immediately when the I/O Signal is raised on the Qualifier pin Thread B begins to execute the rest of the driver.

+ once the device is serviced the execution of thread B will continue, where it could just set up another I/O and yield again.

+ Thread A and B continue to execute in what ever manner the Scheduling Policy dedicates.

MT Thread Register Instructions

- **Dependencies:**
  - VPEConf0[MVP] = 1 (**Master Virtual Processor)**
  - The target TC is identified by the value written to VPEControl[TargTC].
  - A MFTR instruction where the target TC is not in a Halted state, (TCHalt[H] is not set), may result in an UNSTABLE value.
  - If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

MIPS

I am going to go into the Move to and from thread register instructions which provide read/write access to another TC's registers but first let me tell you about how to set up for these instructions.

+ first The VPE that the issuing thread belongs to must be the master Virtual Processor this is done by writing a 1 to the MVP field of the VPEConf0 CP0 register.

+ The target TC is identified by the value in the TargTC field of the VPEControl register.

+ the Target TC should be in a halted state other wise it could result in an unstable value. Most of the time these instructions are used to setup a TC before it is started or restarted so this is usually not a problem.

+ The processor must be in kernel mode and access to Cop 0 must be enabled to execute this instruction.

MT MFTR Instruction

- **Move From Thread Register**
  - **MFTR** is a system coprocessor (Cop0) instruction available to privileged system software for managing thread state.

The MFTR or move from thread register will place the value of the register being read into the GPR referenced by rd in the instruction .

This instruction can be used by an OS to save the context of a TC when a TC needs to be swapped out.

MT MTTR Instruction

- **MoveTo Thread Register**
  - **MTTR** is a system coprocessor (Cop0) instruction available to privileged system software for managing thread state.

There is a corresponding Move to Thread register instruction. The same restrictions and setup that apply to the Move from thread register apply to the Move to thread register instruction

MFTR/MTTR Summary:

- Move value to/from another VPE or TC
- Allows access to GPRs, Hi/Lo, CP0, CP1, CP2 registers
- TC selected by TargTC field of VPEControl CP0 Register
- VPE selected is VPE "containing" the target TC
- Cross-VPE accesses allowed only for "privileged" or "master" VPEs (VPEConf0[MVP] = 1)
- Target TC must be in a Halted state for values to be stable

MIPS 12

In summary of the Move to or from Thread Register instructions

+ They are used to move values to Thread or VPE registers other then the one executing the current code.

+ All CPU register can be accessed

+ The target Thread is selected by setting of the TargTC field in the VPEControl register

+ The VPE is selected is the one the target Thread belongs to this is set in the CurVPE field of the target thread's TCBind register.

+ If you are trying to access VPE registers that are not in the current threads VPE the MVP field in the VPEConf register so before you can access another VPE's registers you first need to set the TargTC in the VPEControl register then make sure the CurVPE field in the Target's TCBind register is set to the other VPE then set the MVP filed of the VPEConf0 register.

+ last the Target thread must be in the halted state.

MT DMT Instruction

- **Disable Multi Threading**
  - **dmt**: suspend all other threads affiliated to the same VPE. This atomically clears the *VPEControl[TE]* bit, returning the original value of *VPEControl* to an optional register argument.
    - It is convenient to bracket a piece of code which needs to be single-threaded within the VPE.

```
dmt rt
... # guaranteed to be the only live TC in this VPE
mtc0 rt, VPEControl
```

MIPS

13

The DMT instruction will disable any other thread that is associated with the VPE the code is executed in.

+ it is a why to insure the currently executing thread executing is the only thread executing in the VPE group. If you have another VPE with threads executing they will still be active.

+ the DMT instruction takes one register as an output. This register will contain the value of the VPEControl register before the DMT took effect.

+ When you want to enable the VPE to its previous state you can use the EMT instruction to enable multi threading but if you are concerned with nested calls it's more robust to replace the whole original value of *VPEControl.*

MT EMT Instruction

- **Enable Multi Threading**
  - The emt instruction atomically sets the *VPEControl[TE]* bit and returns the old value.

The EMT instruction will enable multi threading for the VPE you are executing on.

MT DVPE Instruction

- **Disable Virtual Processing Elements.**
  - DVPE is a privileged Cop0 instructions for disabling multi-VPE operation of a processor. In effect this instruction disables all other threads except for the one that is executing it. It clears the MVPControl[EVP] bit, returning the old value.

Example:

```
dvpe rt
...   # guaranteed to be the only live TC in this CPU
mtc0 rt, MVPControl #restore old settings
```

MIPS

15

The DVPE instruction disables all multithreading, including any other TCs Affiliated to other VPEs, leaving the current thread running alone.

+ the DVPE instruction takes one register as an output. This register will contain the value of the VPEControl register before the DVPE took effect.

+ When you want to enable the CPU to its previous state you can use the EVPE instruction to enable multi threading on all VPEs but if you are concerned with nested calls it's more robust to replace the whole original value of VPEControl register.

MT EVPE Instruction

- **Enable Virtual Processing Elements**
  - The evpe instruction atomically sets the *MVPControl[EVP]* bit and returns the old value.

16

The EVPE instruction enables all VPEs in the CPU.

EMT/DMT, EVPE/DVPE: Summary:

- **Enable/Disable pairs to force serial execution on a single VPE (EMT/DMT) or across all VPEs on a processor (EVPE/DVPE)**
- **Handy for OS critical regions around use of shared resources**
  - Global OS Memory Variables
  - Shared CP0 Resources (e.g. VPEControl)
  - Safe and Easy to Use within Exception Handlers
- **EVPE/DVPE allowed only for "privileged" or "master" VPEs**

MIPS

17

Enable/Disable pairs to force serial execution on a single VPE using the EMT/DMT pair or across all VPEs on a processor using the EVPE/DVPE pair

+ Handy for OS critical regions around use of shared resources

+ Global OS Memory Variables

+ Shared CP0 Resources (e.g. VPEControl)

+ Safe and Easy to Use within Exception Handlers

Note I will also cover later in another section, Inter Thread Communication that can be use for semaphore control which maybe a better way to control shared resources.

+ EVPE/DVPE allowed only for "privileged" or "master" VPEs so the MVP field in the VPEConf register must be set before you can execute this instruction.