The MIPS R4000, part 3: Multiplication, division, and the temperamental HI and LO registers

devblogs.microsoft.com/oldnewthing/20180404-00

April 4, 2018

Raymond Chen

The MIPS R4000 can perform multiplication and division in hardware, but it does so in an unusual way, and this is where the temperamental *HI* and *LO* registers enter the picture.

The *HI* and *LO* registers are 32-bit registers which hold or accumulate the results of a multiplication or addition. You cannot operate on them directly. They are set by a suitable arithmetic operation, and by special instructions for moving values in and out.

The multiplication instructions treat *HI* and *LO* as a logical 64-bit register, where the high-order 32 bits are in the *HI* register and the low-order 32 bits are in the *LO* register.

```
MUL     rd, rs, rt      ; rd = rs * rt, corrupts HI and LO
MULT    rs, rt          ; HI:LO = rs * rt (signed)
MULTU   rs, rt          ; HI:LO = rs * rt (unsigned)
```

The simplest version is `MUL` which multiples two 32-bit registers and stores a 32-bit result into a general-purpose register. As a side effect, it corrupts the *HI* and *LO* registers. (This is the only multiplication or division operation that puts the result in a general-purpose register instead of into *HI* and *LO*.)

The `MULT` instruction multiplies two signed 32-bit values to form a 64-bit result, which it stores in *HI* and *LO*.

The `MULTU` instruction does the same thing, but treats the factors as unsigned.

The next group of multiplication instructions performs accumulation.

```
MADD    rs, rt          ; HI:LO += rs * rt (signed)
MADDU   rs, rt          ; HI:LO += rs * rt (unsigned)
MSUB    rs, rt          ; HI:LO -= rs * rt (signed)
MSUBU   rs, rt          ; HI:LO -= rs * rt (unsigned)
```

After performing the appropriate multiplication operation, the 64-bit result is added to or subtracted from the value currently in the *HI* and *LO* registers.

Note that the `U` suffix applies to the signed-ness of the multiplication, not to whether the operation traps on signed overflow during addition or subtraction. None of the multiplication instructions trap.

The operation runs faster if you put the smaller factor in *rt*, so if you know (or suspect) that one of the values is smaller than the other, you can try to arrange for the smaller number to be in *rt*.

You might think that the division operations take a 64-bit value in *HI* and *LO* and divide it by a 32-bit register. But you'd be wrong. They divide a 32-bit value by another 32-bit value and store the quotient and remainder in in *HI* and *LO*.

```
DIV     rd, rs, rt      ; LO = rs / rt, HI = rs % rt (signed)
DIVU    rd, rs, rt      ; LO = rs / rt, HI = rs % rt (unsigned)
```

None of the division operations trap, not even for overflow or divide-by-zero. If you divide by zero or incur division overflow, the results in *HI* and *LO* are garbage. If you care about overflow or division by zero, you need to check for it explicitly.

Okay, that's great. We've done some calculations and put the results into *HI* and *LO*. But how do we get the answer out? (And how do you put the initial values in, if you are using `MADD` or `MSUB`?)

```
MFHI    rd              ; rd = HI "move from HI"
MFLO    rd              ; rd = LO "move from LO"
MTHI    rs              ; HI = rs "move to HI"
MTLO    rs              ; LO = rs "move to LO"
```

The multiplication and division operations take some time to execute,[1] and if you try to read the results too soon, you will stall until the results are available. Therefore, it's best to distract yourself with some other operations while waiting for the multiplication or division operation to do its thing. (For example, you might check if you need to raise a runtime exception because you just asked the processor to divide by zero.)

The temperamental part of the *HI* and *LO* registers is in how you read the values out.

Tricky rule number one: Once you perform a `MTHI` or `MTLO` instruction, *both* of the previous values in *HI* and *LO* are lost. That means you can't do this:

```
MULT    r1, r2          ; HI:LO = r1 * r2 (signed)
... stuff that doesn't involve HI or LO ...
MTHI    r3              ; HI = r3
... stuff that doesn't involve HI or LO ...
MFLO    r4              ; r4 = GARBAGE
```

You might naïvely think that the `MTHI` replaces the value in the *HI* register and leaves the *LO* register alone, but since this is the first write to either of the *HI* or *LO* registers since the last multiplication or division operation, *both* registers are lost, and your attempt to fetch the value of *LO* will return garbage.

Note that this applies only to the first write to *HI* or *LO*. The second write behaves as you would expect. For example, if you perform `MTHI` followed by `MTLO`, the `MTHI` will set *HI* and corrupt *LO*, but the `MTLO` will set *LO* and leave *HI* alone.

Tricky rule number two: If you try to read a value from *HI* or *LO*, you must wait two instructions before performing any operation that writes to *HI* or *LO*.[2] Otherwise, the reads will produce garbage. The instruction that writes to *HI* or *LO* could be a multiplication or division operation, or it could be `MTHI` or `MTLO`.

Tricky rule number two means that the following sequence is invalid:

```
DIV     r1, r2          ; LO = r1 / r2, HI = r1 % r2 (signed)
... stuff that doesn't involve HI or LO ...
MFHI    r3              ; r3 = r1 % r2 GARBAGE
MULT    r4, r5          ; HI:LO = r4 * r5 (signed)
```

Since the `MULT` comes too soon after the `MFHI`, the `MFHI` will put garbage into *r3*. You need to stick two instructions between the `MFHI` and the `MULT` in order to avoid this.

(Tricky rule number two was removed in the R8000. On the R8000, if you perform a multiplication or division or `MTxx` too soon after a `MFxx`, the processor will stall until the danger window has passed.)

Okay, next time we'll look at constants.

[1] Wikipedia says that latency of 32-bit multiplication was 10 cycles, and latency of 32-bit division was a whopping 69 cycles.

[2] Commenter David Holland explains that this weird rule is due to a pipeline hazard: The multiply or divide operation is not recalled if an exception occurs while the operation is in flight. If the `MFLO` and a subsequent multiply are both in flight and an interrupt occurs, the multiply will complete by the time the exception handler gets around to saving the *HI* and *LO* registers. When execution resumes at the `MFLO`, it will read the low result of the *following* multiplication, rather than the preceding one. That's why you have to wait two cycles: You have to make sure that the `MFLO` has cleared the pipeline before initiating any new operations that may write to *HI* and *LO*.

Raymond Chen

Follow