The MIPS R4000, part 9: Branch delay slot parlor tricks

devblogs.microsoft.com/oldnewthing/20180412-00

April 12, 2018

Raymond Chen

Last time, we learned about the MIPS branch delay slot. Today, we'll look at some tricks you can play with the branch delay slot.

First trick: It is legal to jump into a branch delay slot. Of course, it's not a branch delay slot when you do that. This lets you write some wacky-looking code:

```
    B       somewhere           ; unconditional branch
label:
    OR      v0, zero, zero      ; v0 = 0
...
```

When the unconditional branch is taken, the *v0* register is set to zero before execution continues at the branch destination.

Meanwhile, if somebody jumps to `label`, then execution continues at `label`, which sets *v0* to zero, and then continues with other stuff.

The instruction at `label` acts both as the branch delay slot for the unconditional branch that precedes it, but it's also the first instruction in the basic block if somebody jumps directly into it.

I've seen the opportunity arise for this sort of "squeeze out a single instruction" optimization, but the Microsoft compiler doesn't take advantage of it. Which is probably a good thing. (For one thing, it makes it much harder for binary transformation tools to decompose a program into basic blocks and recombine them in different ways.)

Another branch delay slot trick is editing the return address as part of the jump.

```
    BAL     somewhere
    ADDIU   ra, ra, 4

    NOP

    NOP ; the routine returns here!
```

The `BAL` instruction sets the *ra* register to point to the instruction after the branch delay slot, which in our case is the first `NOP`. But in the branch delay slot, we modify the *ra* register, so that when execution reaches the start of the called procedure, it gets an artificial return address.

I'm told this sort of trick is used by some compilers to combine a call and an unconditional jump into a call with fake return address. For example, in this code fragment

```
    if (...) {
        ...
        function1(...);
    } else {
        ...
    }
    // resume
    x = 0;
```

the call to `function1` is probably followed by an unconditional jump to skip over the `else` branch.

```
    BAL     function1
    NOP                     ; garbage in the branch delay slot
    B       resume
    OR      v0, zero, zero  ; set x = 0

    ... else-branch code goes here ...

    OR      v0, zero, zero  ; set x = 0
resume:
    ...
```

A sneaky compiler could generate the following code:

```
    BAL     function1
    ADDIU   ra, ra, resume - nominal_return ; tweak return address
nominal_return:

    ... else-branch code goes here ...

resume:
    OR      v0, zero, zero  ; set x = 0
    ...
```

In the branch delay slot, we edit the return address so that when `function1` returns, it resumes execution at `resume` rather than `nominal_return`, thereby avoiding having to execute another branch instruction. (We also were able to remove the duplicate `OR v0, zero, zero` instruction that had been hoisted into the branch delay slot of the unconditional branch.) Note that you get this savings only because you had a garbage `NOP` in the branch delay slot. If there were a useful instruction there, then the transformation would go like this:

```
    // original code
    BAL     function1
    MOVE    a0, r0      ; set parameter for function
    B       resume
    OR      v0, zero, zero  ; set x = 0

    // sneaky code
    MOVE    a0, r0      ; set parameter for function
    BAL     function1
    ADDU    ra, ra, ... ; tweak return address

resume:
    OR      v0, zero, zero  ; set x = 0
```

The instruction in the `BAL` instruction's branch delay slot would have to go somewhere else, so you didn't save any time (though you still saved one instruction of space by avoiding duplication of the `OR v0, zero, zero`).

But as we saw earlier, this trick defeats the return address predictor,[1] so it's probably a bad idea.

Okay, next time, we're going to look at the calling convention a bit more closely.

Bonus chatter: Another extra sneaky trick is reusing the return address. Suppose your interpreter loop goes like this:

```
void interpreter_loop(interpreter_state* state)
{
 for (;;) {
  uint32_t opcode = *state->pc;
  state->pc++;
  jump_table[opcode](state, opcode, state->pc);
 }
}
```

The interpreter loop just dispatches to the next opcode forever. Presumably you would break out of this loop with a `longjmp` or some other nonlocal transfer.

The handler function is given the current interpreter state (so it can update it), and as a courtesy, it also gets the current opcode and a pointer to the next unparsed byte as a convenience.

```
interpreter_loop:
    ...
    MOVE    s0, a0       ; s0 points to the interpreter state
    LA      s1, jump_table
    LA      ra, next_opcode ; Footnote ²
next_opcode:
    LW      v1, 80(s0)  ; get address of next opcode byte
    ADDU    a2, v1, 1   ; move to next opcode byte (also argument for handler)
    LBU     a1, 0(v1)   ; load current opcode byte (also argument for handler)
    SW      a2, 80(s0)  ; save pointer to next opcode byte
    SLL     t0, a1, 2   ; multiple by 4 to index jump table
    ADDU    t0, t0, s1  ; calculate entry in jump table
    LW      v0, 0(t0)   ; load the jump target
    JR      v0          ; jump to handler - will return to next_opcode
    MOVE    a0, s0      ; argument for handler
```

When we call the first handler, *ra* is set equal to `next_opcode`. That handler will do its work and then return to the caller by restoring the return address to the *ra* register and performing a `JR ra`.

This means that when control returns to `next_opcode`, you know that *ra* is equal to `next_opcode`! Since that's the value you wanted to be in that register anyway, you can just leave it there when you jump to the next handler, saving you the trouble of having to branch back up to `next_opcode` explicitly.

This seems to be a really clever trick, but it is probably not that useful in practice because of that return address predictor thing.

[1] On the other hand, the MIPS R4000 does not have separate opcodes for "jump indirect to register" and "jump indirect to register for the purpose of returning"; it uses the `JR` instruction for both cases.

The inability to distinguish whether a jump instruction was semantically a return instruction was a non-issue in the original implementation of the MIPS architecture. It had only a two-stage pipeline, so the single branch delay slot was sufficient to avoid ever needing to predict any branches at all.

The MIPS R4000 had a four-stage pipeline, and a branch misprediction would consequently suffer a 2-cycle stall. The MIPS designers codified existing practice and retroactively declared that if the register operand in the `JR` instruction is *ra*, then it predicts as a subroutine return; otherwise it predicts as a computed jump.

[2] For extra sneakiness (and to save an instruction),[3] the loop preparation code could have been written as

```
    LA      s1, jump_table
    BAL     next_opcode
    MOVE    s0, a0       ; s0 points to the interpreter state
next_opcode:
```

This version lets the processor calculate the address of `next_opcode` by performing a `BAL`. This sets the return address to the instruction after the branch delay slot, which is `next_opcode`, and then jumps to… `next_opcode`, which is where the instruction would have gone anyway.

[3] Mind you, this size savings costs you a pipeline stall. See footnote 1.

Raymond Chen

Follow