The MIPS R4000, part 11: More on branch delay slots

devblogs.microsoft.com/oldnewthing/20180416-00

April 16, 2018

Raymond Chen

There seems to be a lot of confusion over branch delay slots. Instead of addressing each comment, I'll just make a post out of it.

The branch delay slot is a dynamic concept. An instruction is in a branch's delay slot if the runtime-determined previously-executed instruction was a branch, regardless of whether the branch was taken.[1]

In casual conversation, however, we tend to talk about "is in a branch delay slot" as a static concept: Would the instruction be in a branch delay slot under the expected sequence of execution?

Let's look at our first example again:

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

Is the `OR` instruction in a branch delay slot?

It depends.

Here's one possibility:

```
    ;   execution starts here
    ADDIU   a0, zero, 4         ; a0 = 4
    B       somewhere           ; unconditional branch
label:
    OR      v0, zero, zero      ; v0 = 0
```

Executing the `B` instruction puts the `OR` instruction is in its branch delay slot.

Here's another possibility:

```
    ;   execution starts here
    J       label               ; unconditional branch
    NOP

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

In this case, the `B` never executed. Therefore the `OR` is not in the branch delay slot of the `B`. It is also not in the branch delay slot of the `J`, because the previously executed instruction was the `NOP` (which was itself in the branch delay slot of the `J`).

This means that it is technically legal to write two branch instructions back to back, provided the first branch is never executed.

```
    J       somewhere
label:
    J       somewhere_else
    NOP
```

If execution of the first `J` never occurs, then there is no violation of the "you cannot put a branch instruction in a branch delay slot" rule, because the second `J` is never in a branch delay slot.

If you think about it, this is how branch delay slots have to be defined in order to make programming tractable and to avoid the processor making spurious memory accesses just to detect whether it's in a branch delay slot.

Suppose a branch delay slot had been defined as "An instruction which has a branch instruction four bytes earlier in memory (whether or not that branch instruction was executed)"; let's call this an *alternate-universe branch delay slot*. You could never start a basic block with a branch instruction, because you don't know what four bytes will come before it in memory. There might be data embedded in the code segment, and the last piece of data might just by pure chance happen to decode as a branch instruction.

```
lookup_table:
    .word   1, 3, 5, 7, 2, 4, 6, 8

trampoline:
    J       actual_destination
    NOP
```

It so happens that the integer 8 is the encoding of the `JR zero` instruction, which is a branch instruction. (Not a very useful branch instruction, but the processor doesn't care about whether your code is useful.) If somebody jumps to the trampoline, it will so happen that the four bytes preceding the `J` instruction form another jump instruction, which would

be a violation of the rule that you cannot put a branch instruction in an alternate-universe branch delay slot. The compiler would have to insert a `NOP` at the start of every trampoline to ensure that the `J` does not lie in an alternate-universe branch delay slot.

Furthermore, it means that after every branch instruction, the processor would have to fetch *two* instructions from memory, one for the instruction being executed, and the other (four bytes earlier) to determine whether the instruction is in an alternate-universe branch delay slot.

And if the previous four bytes are (heaven forfend) on a not-present page, the processor would have to raise a page fault in order to get the operating system to cough up those four preceding bytes. (And if the previous page were invalid, then um I don't know what you would have to do.)

Thankfully, that's not how the rule is written. Branch delay slots are determined at run time based on instructions actually executed.

Okay, so what happens if you put a branch in a branch delay slot?

First of all, the primary purpose of this series is to help you debug user-mode MIPS assembly code, and since no well-formed MIPS assembly code would put a branch in a branch delay slot, this is not something you would ever encounter when debugging, so the question is technically out of scope.

But let's try to answer it anyway.

The processor architecture officially says that the result of putting a branch in a branch delay slot is **UNPREDICTABLE**, which is a technical term that means, basically, "Anything can happen, but limited to things that the code could already do at its current privilege level." So it might scramble all your registers to nonsense values, or fill them with the contents of randomly-selected memory addresses, provided your privilege level has access to that memory. It could jump to an arbitrary location. It could raise an exception. But it cannot, say, cause user-mode code to load data from kernel-only memory space, or hang the processor.

So the answer to "What happens?" is "I can't answer that because it's not defined."

But let's try to answer it anyway.

On some versions of the MIPS processor, it will raise an invalid instruction exception.

On other versions of the MIPS processor, it will try to execute the branch anyway.

So let's pick a specific processor that tries to execute the branch, say one of the original MIPS processors with a two-stage pipeline.

You can model the processor like this:

```
int program_counter;
bool in_branch_delay_slot;
std::array<pipeline, 2> unit;

while (true) {
 // Fetch and decode the next instruction.
 unit[0].fetch_instruction(program_counter);

 program_counter += 4;

 // Execute the previous instruction.
 if (unit[1].is_branch_instruction()) {
   program_counter = unit[1].calculate_branch_target();
 } else {
   ... handle other types of instructions ...
 }

 // Remember whether the previous instruction was a branch.
 in_branch_delay_slot = unit[1].is_branch_instruction();

 // Advance the pipeline
 std::rotate(unit.first(), unit.last() - 1, unit.last());
}
```

In reality, the "Fetch and decode" and the "Execute the instruction" steps occur in parallel, but we do it sequentially here for expository purposes.

Let's step through a normal code sequence that involves a branch instruction:

```
10000000:   B   10000020
10000004:   LW  v0, 80(t0)

10000020:   ADD v0, v1, v0
```

We start with `program_ counter = 10000000`.

We fetch the branch instruction into unit zero, advance the instruction pointer to `10000004`, and then finish whatever the previous instruction was. Assume that the previous instruction was not a branch instruction, so `in_ branch_ delay_ slot` is false. Finally, we advance the pipeline, so that the work that was previously in unit zero will continue in unit one. (That's not really how processors work, but this is a model, not the real thing.)

At the next cycle, `program_ counter = 10000004`.

We fetch the `LW` instruction into unit zero, advance the instruction pointer to `10000008`, and then execute the branch instruction in unit one, which means that the `program_ counter` is changed to `10000020` and `in_ branch_ delay_ slot` is now

true. That's the end of this cycle, so we advance the pipeline again.

At the next cycle, `program_ counter = 10000020`.

We fetch the `ADD` instruction into unit zero, advance the instruction pointer to `10000024`, and then execute the `LW` instruction in unit one to load some memory into a register. This was not a branch instruction, so `in_ branch_ delay_ slot` is now false. And then we advance the pipeline.

This shows more concretely why the processor has a branch delay slot: The instruction after the branch is already in the pipeline, so it will finish executing. The branch controls what enters the pipeline *next*. Since we have a two-stage pipeline, that means that the effect of the branch isn't visible until two instructions later.

Suppose there is an exception at the `LW`, say, because the page was not present.

When an exception occurs, the processor captures the address of the faulting instruction into a special control register called *EPC* (Exception Program Counter), and the value of `in_ branch_ delay_ slot` is captured into a special control flag called *BD* (Branch Delay). The kernel trap handler copies the values out of these control registers so it can resume execution after handling the exception.

In our case, the kernel receives a TLB Invalid exception to say "Hey, somebody tried to access invalid memory." The processor is kind enough to provide the address that was invalid (in this case `80 + t0`) so the kernel doesn't have to try to parse the faulting instruction to figure out the address.

The kernel does whatever it needs to do to make the memory present, updates the TLB, and then it's ready to resume execution.

The processor helps you out a little here: It does the work of backing up the instruction pointer by four bytes if the *BD* flag is set.[2] In other words, if the faulting instruction was in a branch delay slot, then the value in *EPC* is the address of the faulting instruction *minus four*.

To resume execution after handling the exception, the kernel just needs to restore the processor registers, and then jump to *EPC*.

In our case, it means that when the exception is raised at the `LW`, the captured values are *BD* = true, *EPC* = `10000004 - 4 = 10000000`.

Okay, so now let's do something crazy: Let's put a branch instruction in a branch delay slot.

```
20000000:   B   20000020
20000004:   B   20000040

20000020:   LW  v0, 80(t0)

20000040:   ADD v0, v1, v0
```

We start with `program_ counter = 20000000`.

We fetch the first branch instruction into unit zero, advance the instruction pointer to `20000004`, and then finish whatever the previous instruction was. That's the end of this cycle, so we advance the pipeline.

At the next cycle, `program_ counter = 20000004`.

We fetch the second branch instruction into unit zero, advance the instruction pointer to `20000008`, and then execute the first branch instruction in unit one, which means that the `program_ counter` is changed to `20000020`, and `in_ branch_ delay_ slot` is now true. That's the end of this cycle, so we advance the pipeline again.

At the next cycle, `program_ counter = 20000020`.

We fetch the `LW` instruction into unit zero, advance the instruction pointer to `20000024`, and then execute the second branch instruction in unit one, which means that the `program_ counter` is changed to `20000040`, and `in_ branch_ delay_ slot` is still true. That's the end of this cycle, so we advance the pipeline again.

At the next cycle, `program_ counter = 20000040`.

We fetch the `ADD` instruction into unit zero, advance the instruction pointer to `20000044`, and then execute the `LW` instruction in unit one. Oh no, this instruction takes a page fault!

The processor captures the current value of the `in_ branch_ delay_ slot` flag (true) into the *BD* special control register, and it captures the address of the faulting instruction (`20000020`) into the *EPC* special control register, And since the *BD* flag is set, the processor subtracts four from `20000020`, leaving `2000001C`.

The kernel processes the page fault by paging in the necessary data, and then it resumes execution at *EPC*, which is `2000001C`. It's resuming execution at an instruction that wasn't part of the original instruction stream!

So that's one possible result of putting a branch instruction in a branch delay slot: From the user-mode code's point of view, the CPU lost its mind and jumped to the wrong address.

I reiterate that this is just one possible result. The result of putting a branch instruction in a branch delay slot is architecturally **UNPREDICTABLE**, so what actually happens is anybody's guess.[3]

Next question: What happens if you jump into your own branch delay slot?

The instruction in the branch delay slot executes twice. It executes once because it's in the branch delay slot. It executes again because it's the destination of the branch.

One final note is the case of emulated instructions. For example, maybe it was a misaligned memory access, or it was a floating point operation on a system with no floating point coprocessor. In the cases of emulation, the kernel wants to step over the emulated instruction and resume execution at the next instruction. But what if the emulated instruction was in a branch delay slot?

The kernel detects that it is in the ugly case by observing that the *BD* flag is set. In that case, the kernel must back up and emulate the branch instruction, too! The kernel determines whether the branch was taken or not-taken by inspecting the instruction opcode and the contents of the relevant registers, and it resumes execution at the appropriate instruction: Either the branch target if the branch was taken, or the instruction after the delay slot if the branch was not taken. As we noted earlier, the processor already updated the return address register if applicable, so at least the kernel doesn't need to emulate that part of the instruction.

[1] Note that the SPIM emulator gets this wrong. It sets the `running_in_delay_slot` variable to `1` only for taken branches.

[2] The flip side of this behavior is that if you want to identify the faulting instruction, you have to add four to *EPC* if the *BD* flag is set.

[3] Indeed, the SPIM emulator does a third thing: It executes the instruction that comes sequentially after the second branch instruction, and then continues execution at the destination of the first jump instruction.

Raymond Chen

Follow