The MIPS R4000, part 15: Code walkthrough

devblogs.microsoft.com/oldnewthing/20180420-00

April 20, 2018

Raymond Chen

Today we're going to take a relatively small function and watch what the compiler did with it. The function is this guy from the C runtime library, although I've simplified it a bit to avoid some distractions.

```
extern FILE _iob[];

int fclose(FILE *stream)
{
    int result = EOF;

    if (stream->_flag & _IOSTRG) {
        stream->_flag = 0;
    } else {
        int index = stream - _iob;
        _lock_str(index);
        result = _fclose_lk(stream);
        _unlock_str(index);
    }

    return result;
}
```

Here's the corresponding disassembly:

```
; int fclose(FILE *stream)
; {
    addiu   sp,sp,-0x28     ; reserve stack space
    sw      ra,0x1C(sp)     ; save return address
    sw      s0,0x18(sp)     ; save s0
```

On entry, the parameters to a function are passed in *a0* through *a3*. This function has only one parameter, so it goes in *a0*.

We reserve some stack space. The first 16 bytes of that stack space are going to be used as home space for the functions we call, so our usable bytes start at offset `0x10`. We save the *s0* register (because we're going to use it as a local variable) and the return address (because it will be modified when we call to other functions).

```
;   if (stream->_flag ...
    lw      t6,0xc(a0)      ; t6 = stream->_flag
    move    a1,a0           ; save stream in a1
```

We are going to test a bit in the `stream->_flag` member, so we need to load that up. Meanwhile, we save the `stream` parameter in the *a1* register.

```
;   int result = EOF;
    li      v1,-1           ; result = -1
```

Interleaved with the evaluation of the condition we insert the initialization of the `result` local variable.

```
;   if (stream->_flag & _IOSTRG) {
    andi    t7,t6,0x40      ; is the _IOSTRG bit set?
    bnezl   t7,done         ; yup, then bail

;       stream->_flag = 0;
    sw      zero,12(a0)     ; but set stream->_flag = 0 before we go
```

We mask off all but the `_IOSTRG` bit and see if it's nonzero. If so, then we branch. This branch uses the `l` "likely" suffix, so the instruction in the branch delay slot executes only if the branch is taken. Since the `true` branch of the `if` is only one instruction long, the entire contents fit inside the delay slot. How convenient. We can put the `true` branch in the branch delay slot and jump right to the function exit code. If the branch is not taken, then the instruction in the branch delay slot is suppressed. (This suppression behavior is the case only for `l`-type branches.)

```
;   } else {
;       int index = stream - _iob;
    lui     t8,0x77cd       ; load address of _iob into t8
    addiu   t8,t8,0x7b0     ; t8 = 0x77cd07b0
    subu    s0,a1,t8        ; calculate raw pointer offset
    sra     s0,s0,5         ; divide by 32 to get the index (saved in s0)
```

To calculate the pointer difference, we need to subtract the raw pointers, and in order to do that, we need to load the 32-bit address of the `_iob` array. That takes two instructions. And then we subtract the raw pointers to get the byte difference. And then we divide by `sizeof(FILE)` to get the index. We're lucky that the size of a `FILE` is a power of 2, so a shift instruction can be used instead of a full division.

```
;       _lock_str(index);
    move    a0,s0           ; Load argument for _lock_str
    jal     _lock_str
    sw      a1,0x28(sp)     ; save stream pointer on the stack for later
```

Now that we've calculated the index, set it up as the argument for the `_lock_str` function and call it. But just before we go, we save *a1* (which is the `stream` parameter) on the stack so we don't lose it. The saving of *a1* goes into the branch delay slot, so it executes before the branch is taken, even though it comes after the branch in the instruction stream.

(I don't know why the compiler bothered with *a1*. It could have saved *a0* on the stack sooner and put the `move a0, s0` in the branch delay slot.)

```
;       result = _fclose_lk(stream);
    jal     _fclose_lk
    lw      a0,0x28(sp)     ; load argument for _fclose_lk
```

The next thing to do is to call `_fclose_lk`, and in this case, we load its argument in the branch delay slot. Seeing work happen in the branch delay slot takes getting used to. It always takes a period of adjustment whenever I switch to MIPS after working with some other processor without branch delay slots.

```
;       _unlock_str(index);
    move    a0,s0           ; Load argument for _unlock_str
    jal     _unlock_str
    sw      v0,0x24(sp)     ; result = return value from _fclose_lk
```

After the `_fclose_lk`, we call `_unlock_str`, and this time we use the branch delay slot to save the return value from `_fclose_lk` onto the stack before we lose it. (Though the compiler could have done a little better and saved it in *s0*, since `index` is a dead variable at this point.)

```
;  }
    lw      v1,0x24(sp)     ; recover result so we can return it
```

After `_unlock_str` returns, we put `result` into *v1* because that's where our cleanup code expects it.

Note that in the instruction stream, you see a store immediately followed by a load from the same location. This makes no sense at first, until you realize that there's a function call in between them, because the store is in the branch delay slot. Even though the store and load immediately follow each other in the instruction stream, there's an entire function call that happens in between! The store happens before the function call, ad the load happens after.

```
;  return result;
; }
done:
    move    v0,v1           ; set return value
    lw      s0,0x18(sp)     ; restore s0
    lw      ra,0x1C(sp)     ; recover return address
    jr      ra              ; return
    addiu   sp,sp,0x28      ; clean up stack
```

We set the return value to the `result`, and then we enter the epilogue. In the epilogue, we restore the *s0* register we had been using to hold `index`, and then we load up the return address and jump back to it. We destroy the stack frame in the branch delay slot.

Overall, it's pretty straightforward code. The only truly weird thing is the branch delay slot.

But that's a huge truly weird thing.

This concludes our tour of the MIPS R4000 processor. I never had to do any significant work with it, so I probably won't be able to answer interesting questions. The focus was on learning enough to be able to read valid compiler output, with a few extra notes on the architecture to call out what makes it different.[1]

Bonus chatter: Here's my hand-optimized version of the function.

```
; int fclose(FILE *stream)
; {
    addiu   sp,sp,-0x10     ; reserve stack space
    sw      ra,0x14(sp)     ; save return address
    sw      s0,0x10(sp)     ; save s0
    move    s0,a0           ; register variable s0 = stream
```

My first trick is to reuse the home space. The compiler-generated version didn't use the home space for anything other than saving the `stream` parameter. Look, people, it's free memory! We need three words of stack, one for the return address, one to save the preserved register *s0*, and one to save the index. We get four words of home space, so we can just use that. The actual stack frame needed by our function is just the home space for the outbound call.

(I wonder whether it's legal to overlap your inbound home space with your outbound home space. If our function had needed only two words of stack, would it have been okay for us to write `addiu sp, sp, -8`?)

```
;   if (stream->_flag ...
    lw      t6,0xc(a0)      ; t6 = stream->_flag
;   int result = EOF;
    li      v0,-1           ; result = -1 (avoid load stall)
```

I'm precalculating the `result` in anticipation of the early-out. This instruction is basically free because it comes in the load delay slot. If we had tried to use the value in *t6* immediately, the processor would have stalled for a cycle, so we may as well use that cycle productively, even if only speculatively.

```
;   if (stream->_flag & _IOSTRG) {
    andi    t7,t6,0x40      ; is the _IOSTRG bit set?
    bnezl   t7,done         ; yup, then bail
;       stream->_flag = 0;
    sw      zero,12(a0)     ; but set stream->_flag = 0 before we go
```

The test and one-line-body for the `_IOSTRG` test hasn't changed, except that we exit with the return value directly in *v0* rather than in *v1*.

```
;   } else {
;       int index = stream - _iob;
    lui     t8,0x77cd       ; load address of _iob into t8
    addiu   t8,t8,0x7b0     ; t8 = 0x77cd07b0
    subu    a0,a0,t8        ; calculate raw pointer offset
    sra     a0,a0,5         ; divide by 32 to get the index (in a0)
```

The calculation of the index is the same, except that I put it directly into *a0* so it is ready to be passed to `_lock_str`.

```
;       _lock_str(index);
    jal     _lock_str
    sw      a0,0x18(sp)     ; save index
```

I spent some time trying to decide which should be the register variable: `stream` or `index`. Turns out it doesn't matter from a code size point of view: Both are saved and restored exactly once.

```
;       result = _fclose_lk(stream);
    jal     _fclose_lk
    move    a0,s0           ; load argument for _fclose_lk
```

Calling `_fclose_lk` is simpler because we can move the argument from a register rather than from memory. That way, if the first thing that `_fclose_lk` does is try to use the stream, it won't suffer a load delay stall. The first instruction of the called function executes immediately after the branch delay slot. If you put a load instruction in the branch delay slot, then the first instruction of the called function is executing in a load delay slot, and it probably isn't expecting that.

So that thinking tipped the scales in favor of keeping `stream` as the register variable. (Of course, that thinking is also based on the older MIPS implementation, which was not dual-issue. The MIPS R4000 processes one instruction every half-cycle. This alters the micro-optimization considerations for both branch delays and load delays.)

```
;       _unlock_str(index);
    lw      a0,0x18(sp)     ; Load argument for _unlock_str
    jal     _unlock_str
    move    s0,v0           ; result = return value from _fclose_lk
```

We could have swapped the `lw` and `move`, but I load early and move late in order to avoid loading memory in a branch delay slot, for reasons explained above.

Since the `stream` variable is dead, we can reuse the *s0* register to hold `result`.

```
;  }
    move    v0,s0           ; recover result so we can return it
;  return result;
; }
done:
    lw      s0,0x10(sp)     ; restore s0
    lw      ra,0x14(sp)     ; recover return address
    jr      ra              ; return
    addiu   sp,sp,0x10      ; clean up stack
```

And then we clean up and go home. Everybody reached `done` with the return value already in `v0`, so all that's left to do is restore our registers and stack.

[1] I confess that the excursion into branch delay slots took me away from the focus on how to read valid compiler output. Sorry.

Raymond Chen

Follow