The PowerPC 600 series, part 14: Code walkthrough

devblogs.microsoft.com/oldnewthing/20180823-00

August 23, 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)
; {
    mflr    r0              ; move return address to r0
    stw     r29,-0xC(r1)    ; save non-volatile register
    stw     r30,-8(r1)      ; save non-volatile register
    stw     r31,-4(r1)      ; save non-volatile register
    stw     r0,-0x10(r1)    ; save return address
    stwu    r1,-0x50(r1)    ; create stack frame and link
```

On entry, the parameters to a function are passed in *r3* through *r10*. This function has only one parameter, so it goes in *r3*.

The return address is passed in *lr*, but the *lr* register cannot be stored directly into memory. We need to transfer it through a general-purpose register. The Microsoft compiler uses *r0* for this purpose (and doesn't use *r0* for any other purpose as far as I can tell.)

The next step is to save the non-volatile registers that the function uses, so that they can be restored at function exit. Then we save the return address on the stack, and finally create the stack frame and link it to the previous stack frame.

We created an 80-byte stack frame. The 24 bytes closest to the top of the stack form the system-reserved area; the next 32 bytes are the home spaces for the eight register parameters. We don't call any functions with more than eight parameters, so we don't need any space for the outbound parameters beyond eight. Our usable local variables therefore start at offset 56. On the other hand, we stored the return address at offset 80 – 16 = 64, and the nonvolatile registers at offsets 68 thorough 76, which means that our local variables live at offsets 56 through 64. (It turns out that we won't use any of them! But we had to allocate them anyway, in order to keep the stack aligned on a 16-byte boundary.)

Okay, with the prologue out of the way, we can start doing real work.

```
;   if (stream->_flag ...
    mr      r31,r3          ; r31 = stream
    lwz     r3,0xC(r31)     ; r3 = stream->_flag
```

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 *r31* register.

```
;   int result = EOF;
    li      r30,-1          ; r30 = -1
    li      r4,0            ; r4 = 0 (handy zero value)
```

Interleaved with the evaluation of the condition we insert the initialization of the `result` local variable, and we set *r4* to zero because zero is a handy value to have.

```
;   if (stream->_flag & _IOSTRG) {
    rlwinm. r3,r3,0,25,25   ; r3 = r3 & 0x40 (_IOSTRG)
    beq     notstring       ; if bit not set, then go to "else" branch
```

We use the all-purpose `rlwinm` instruction here. We shift by zero positions, but specify a mask of (25,25). On the PowerPC, bits are numbered starting from the most significant bit, so position 25 has value `1 << (31-25) = 0x40`. Therefore, this instruction is functionally equivalent to

```
    andi.   r3,r3,0x40      ; r3 = r3 & 0x40 (_IOSTRG)
```

Since the `rlwinm` opcode is followed by a period, it sets flags in *cr0* based on the result. We test these flags in the subsequent `beq` and jump if the bit is not set. Recall that if you don't specify a condition register for `beq`, it defaults to *cr0*.

Otherwise, we fall through:

```
;       stream->_flag = 0;
    stw     r4,0xC(r31)     ; stream->_flag = 0
    b       done            ; end of "true" branch
```

We preloaded zero into the *r4* register, so we can use a `stw` to store that zero into the `stream->_flags`. That's the end of the `true` branch of the `if` statement, so we jump to the function exit code.

```
    } else {
notstring:
;       int index = stream - _iob;
    lwz     r3,-0x7F3C(r2)  ; r3 = &_iob
    subfc   r3,r3,r31       ; calculate raw pointer offset
    srawi   r29,r3,5        ; divide by 32 to get the index (saved in r29)
```

First, we need to calculate the address of the `_iob` global address. The addresses of global variables are kept in the table of contents. The displacement in memory access instructions is a signed 16-bit value, so the table of contents register usually points 32KB past the start of the actual table of contents, so that the code can use both positive and negative offsets to access a 64KB block of data. And since most programs don't have more than 8192 global variables, the offsets you see will almost always be negative.

After we get the address of the `_iob` global variable, 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);
    mr      r3,r29          ; first function parameter is "index"
    bl      _lock_str       ; call _lock_str
    nop                     ; don't need to restore toc
```

Now that we've calculated the index, set it up as the argument for the `_lock_str` function and call it. At the time the compiler generated the code, it was not sure whether `_lock_str` was a function in the same module or was a naïvely-imported function, so it left a `nop` after the `bl`. If the function turned out to be a naïvely-imported function, the linker would have changed the `nop` to `lwz r2, 4(sp)` in order to restore the table of contents.

```
;       result = _fclose_lk(stream);
    mr      r3,r31          ; load parameter for _fclose_lk
    bl      _fclose_lk
    mr      r30,r3          ; save return value in "result"
```

The next thing to do is to call `_fclose_lk`, so we put the `stream` parameter in *r3*, copying it from *r31* which is where we saved it at the start of the function. This time, the compiler knows that `_fclose_lk` is in the same module, presumably because it was in the same translation unit, so it doesn't need to leave a `nop` after the `bl`.

```
;       _unlock_str(index);
    mr      r3,r29          ; load parameter for _unlock_str
    bl      _unlock_str
    nop                     ; don't need to restore toc
```

After the `_fclose_lk`, we call `_unlock_str`, and this time the compiler didn't know whether `_unlock_str` was in the same module or not, so it leaves a precautionary `nop` after the `bl`.

```
;  }
done:
    mr      r3,r30          ; set return value

    lwz     r0,0x40(r1)     ; recover return address
    lwz     r29,0x44(r1)    ; restore non-volatile register
    lwz     r30,0x48(r1)    ; restore non-volatile register
    lwz     r31,0x4C(r1)    ; restore non-volatile register
    mtlr    r0              ; move return address to lr so we can jump to it
    addi    r1,r1,0x50      ; clean the stack
    blr                     ; return to caller
```

We set the return value to the `result`, and then we enter the epilogue. In the epilogue, we load the return address into *r0*, and then restore the non-volatile registers. We load the return address first so that the `mtlr` is less likely to stall waiting for the answer to come back from memory.

One thing you may notice is that the non-volatile registers are saved with negative offsets (into the red zone) but restored from positive offsets (from the local frame). This makes it harder to match up the two, but you can generally assume that the compiler knows how to do math and didn't mess that up.

The more significant consequence of this is that it's harder to manually unwind the stack in order to see what was in the registers of the caller. You can disassemble at the start of the function to see where the registers were saved, but they are saved at negative offsets, which you then need to mentally add to the size of the stack frame expressed in the `stwu` instruction at the end of the prologue. To get the positive offsets, you need to disassemble at the end of the function, which is harder to find since you just keep disassembling forward until you find that you've started disassembling another function. And even that trick doesn't work if the module has undergone profile-guided optimization, which can make the code for a function discontiguous.

Anyway, after restoring the non-volatile registers, we move the return address into the *lr* register, pop the stack frame, and return. (The Windows NT software conventions require that the return instruction be encoded exactly as `blr` and not one of its functional equivalents.)

This concludes our very quick tour of the PowerPC 600 series of processors. Like the MIPS R4000, I never had to do any significant work with PowerPC, 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.

Raymond Chen

Follow