The SuperH-3, part 15: Code walkthough

devblogs.microsoft.com/oldnewthing/20190823-00

August 23, 2019

Raymond Chen

Once again, we wrap up our processor retrospective series by walking through a simple function from the C runtime library.

```
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)
; {
        mov.l   r8,@-r15        ; push r8
        mov.l   r9,@-r15        ; push r9
        mov.l   r10,@-r15       ; push r10
        sts.l   pr,@-r15        ; save return address

        add     #-16,r15        ; allocate space for outbound calls
```

We start by saving the nonvolatile registers that we are going to be using as local variables in this function. Next, we allocate space on the stack to act as home space for our outbound calls. Most function start this way.

```
        mov     r4,r9           ; r9 = stream
```

This function enregisters the `stream` parameter, so save it from the volatile *r4* register into a non-volatile register *r9*. Other register variables are going to be *r10* for `result` and *r8* for `index`.

```
;   int result = EOF;
;
;   if (stream->_flag & _IOSTRG) {

        mov.l   @(12,r9),r3    ; r3 = stream->_flag
        mov     #64,r2         ; r2 = _IOSTRG
        and     r2,r3          ; r3 = stream->_flag & _IOSTRG
        tst     r3,r3          ; is it zero?
        bt/s    isfile         ; Y: so it's a file
        mov     #-1,r10        ; Set r10 = EOF
```

To test the flag, we load the value into a register (*r3*), load the constant `0x40` into another register so we can `AND` them together and test the result. The `TST` instruction implicitly tests against zero, so a *branch if true* means *branch if zero*. If the result is indeed zero, then we branch to the string handling case, but not before setting `r10` to `-1`, which initializes the `result` variable.

```
;       stream->_flag = 0;
;   }

        mov     #0,r3          ; prepare to store zero
        bra     done           ; and we're done
        mov.l   r3,@(12,r9)    ; stream->_flag = 0
                               ; (in the branch delay slot)
```

If we have a string, then we set `_flag` to 0 by loading the constant zero into a register and storing it. Then we jump to the common exit code.

```
;   } else {
;       int index = stream - _iob;

isfile:
        mov.l   @(42,pc),r2 ; #0x10004080 ; load constant address of _iob
        mov     r9,r8          ; r8 = stream
        mov     #-5,r3         ; prepare to shift right 5 places
        sub     r2,r8          ; r8 = stream - _iob (byte offset)
        shad    r3,r8          ; index = stream - _iob (element offset)
```

The `FILE` structure is a convenient 32 bytes in size, so the byte offset can be converted to an element offset by a simple shift. There is no right-shift-by-5 instruction, so we have to do a variable shift. There is no right-shift-by-variable instruction, so we instead do a left shift by the negative, because the left-shift instruction `SHAD` can shift both left *or* right, depending on the sign of the shift amount.

```
;       _lock_str(index);
`

        mov.l   @(36,pc),r3 ; #0x10001040 ; address of _lock_str
        jsr     @r3         ; call it
        mov     r8,r4       ; copy parameter from r8 = index
```

To call the `_lock_str` function, we put the `index` parameter in *r4* (in the delay slot), load up the address of the function, and then call it.

```
;       result = _fclose_lk(stream);
`

        mov.l   @(36,pc),r3 ; #0x10002130 ; address of _fclose_lk
        jsr     @r3         ; call it
        mov     r9,r4       ; copy parameter from r9 = stream
```

And another function call. Note that the displacement for the `@(36,pc)` is the same offset as the previous one, yet it loads a different value. That's because *pc* has changed!

```
;       _unlock_str(index);

        mov.l   @(32,pc),r3 ; #0x100010c8 ; address of _unlock_str
        mov     r8,r4      ; copy parameter from r8 = index
        jsr     @r3        ; call it
        mov     r0,r10     ; save return value of _fclose_lk into result
```

And then call `_unlock_str`. This time, we also have to save the return value from `_fclose_lk` so we can return it from the function.

```
;   }
;   return result;
; }

done:
        add     #16,r15    ; clean the stack
        mov     r10,r0     ; put return value into r0 register
        lds.l   @r15+,pr   ; pop return address
        mov.l   @r15+,r10  ; pop r10
        mov.l   @r15+,r9   ; pop r9
        rts                ; return to caller
        mov.l   @r15+,r8   ; pop r8
```

And we reach the function exit. We put the return value in the *r0* register, because that's what the calling convention dictates. And we undo the stack operations we performed in the function prologue: Clean the stack and pop off the registers.

But wait, we're not done yet. We have those constants in the code segment that we need to generate.

```
        .data.l     _iob
        .data.l     _lock_str
        .data.l     _fclose_lk
        .data.l     _unlock_str
```

When you look at the disassembly, these data bytes are going to be disassembled as if they were code, because the disassembler doesn't know that they're actually data. You just have to understand that nonsense instructions after an unconditional branch are likely to be data.

Bonus chatter: Here's my attempt to hand-optimize the assembly.

First observation is that enregistering a variable that is used only once costs the same as spilling it. If you spill it, you write it to memory once and load it from memory once. If you enregister it, you write the original register to memory once, and restore it from memory once. Either way, you perform one read and one write. This means that the `stream` variable may as well be spilled.

Second observation is that there is really only one interesting live variable across each of the calls. Either we are saving the index, or saving the result. So we can use the same register to hold both.

And the third observation is that the compiler didn't take advantage of the free home space.

```
        mov.l   r8,@(12,r15)    ; save r8 in parameter 4 home space
        sts.l   pr,@(8,r15)     ; save pr in parameter 3 home space
        mov.l   r4,@(4,r15)     ; save stream in parameter 2 home space
```

I have 16 bytes of free memory, so I use it instead of pushing values onto the stack. I used 12 bytes of my home space, so I need to allocate 12 bytes of stack to get myself back up to 16 bytes of home space for the outbound function calls. I'll interleave that with the next sequence of instructions to try to avoid a load stall.

```
        mov.l   @(12,r4),r3     ; r3 = stream->_flag
        add     #-12,r15        ; allocate space for outbound calls
        mov     #64,r2          ; r2 = _IOSTRG
        and     r2,r3           ; r3 = stream->_flag & _IOSTRG
        tst     r3,r3           ; is it zero?
        mov     #-1,r0          ; return value is EOF (if it's a string)
        bf      isstring        ; N: so it's a string
```

The code to test the flag hasn't really changed, but I moved the stack pointer adjustment into this sequence to avoid the stall that occurs when we try to use *r3* too soon after loading it from memory. This delay of the stack pointer adjustment is legal because we are allowed to advance instructions into the prologue provided they are not jumps and do not modify nonvolatile registers.

There is a stall between the `TST` and the `BF` because we are consuming flags immediately after generating them, so I slip a `MOV` instruction in there. The value is used only if the branch is taken, but it does no harm in the fallthrough case, and we may as well try it, since it's a free instruction due to the stall.

```
;       int index = stream - _iob;
;       _lock_str(index);

        mov.l   #_iob,r2         ; r2 = address of _iob
        mov     r4,r8            ; r8 = stream
        mov.l   #_lock_str,r0    ; address of _lock_str
        mov     #-5,r3           ; prepare to shift right 5 places
        sub     r2,r8            ; r8 = stream - _iob (byte offset)
        shad    r3,r8            ; index = stream - _iob (element offset)
        jsr     @r0              ; call _lock_str
        mov     r8,r4            ; copy parameter from r8 = index
```

The code to calculate the index hasn't really changed, but I interleave it with the preparation to call `_lock_str` to avoid a load stall.

```
;       result = _fclose_lk(stream);
`
        mov.l   #_fclose_lk,r3   ; address of _fclose_lk
        jsr     @r3              ; call it
        mov     @(20,r15),r4     ; parameter 1 is the stream
```

This is the same as before, except we load the stream from memory because we didn't dedicate a register to it. This does mean that if the `_fclose_lk` function tries to access its parameter within its first two instructions, it will suffer a load stall. (Normally, we'd have to count four instructions, but there is a one-cycle pipeline bubble on a taken branch, so that sucks up two of the instructions.) However, `_fclose_lk` is almost certainly going to have at least one register variable, so those first two instructions are going to be occupied by spilling *r8* and *pr*. The earliest it is likely to access *r4* is its third instruction, so we're safe.

```
;       _unlock_str(index);

        mov.l   #_unlock_str,r3 ; address of _unlock_str
        mov     r8,r4            ; copy parameter from r8 = index
        jsr     @r3              ; call it
        mov     r0,r8            ; save return value of _fclose_lk into r8
```

The trick here is that the `result` variable becomes live at the same moment that `index` becomes dead, so we can use the same register *r8* for both of them. After the function returns, we put the saved value back into *r0* so we can return it.

```
        bra     done             ; to common exit code
        mov     r8,r0            ; put result back into r0 so we can return it
```

After `_unlock_str` returns, we go to our common exit code, with the desired return value in *r0*.

```
;   int result = EOF;
;   stream->_flag = 0;

isstring:
        mov      #0,r1          ; value to store into stream->_flag
        mov      r1,@(12,r4)    ; stream->_flag = 0
                                ; r0 is already -1
```

In the string case, we just zero out the `_flag` and return `-1`, which we preloaded into *r0* prior to the branch into this code path. Then we fall through to the common exit code.

```
done:
        lds.l   @(20,r15),pr    ; recover return address
        add     #12,r15         ; clean the stack
        rts                     ; return to caller
        mov.l   @(12,r15),r8    ; restore r8
```

And we're done. Our epilogue code is rather brief because we already put the desired return value in the *r0* register, and because we didn't have a lot of saved registers to restore. I put the `add` after the `lds.l` because I'm going to stall on the load delay, so I may as well get a free instruction out of it.

Raymond Chen

Follow