The SuperH-3, part 14: Patterns for function calls

devblogs.microsoft.com/oldnewthing/20190822-00

August 22, 2019

Raymond Chen

Function calls on the SH-3 are rather cumbersome. The BSR instruction has a reach of only 4KB, which makes it impractical for compiler-generated code because the compiler doesn't know where the linker is going to put the function it's calling. In practice, all function calls in compiler-generated code are performed with the `JSR` instruction, which calls a function whose address is given by a register.

The typical case of a direct function call goes like this:

```
MOV.L   r3, @(16, r15)          ; parameter 5 passed on the stack
MOV     r8, r7                  ; parameter 4 copied from another register
MOV     #20, r6                 ; parameter 3 is address of local variable
ADD     r15, r6                 ; r6 = r15 + 20
MOV     #8, r5                  ; parameter 2 is calculated in place
MOV.L   #function, r0           ; r0 = function to call
JSR     @r0                     ; call the function
MOV     @(24,r15), r4           ; parameter 1 copied from the stack
                                ; (in the branch delay slot)
```

We load the function address into some register. The compiler usually uses one of the non-parameter scratch registers for this purpose, *r0* through *r3*. Note that we wrote this as a 32-bit immediate, but that is a pseudo-instruction which the assembler converts to a PC-relative load, with a constant embedded in the code segment.

```
; You write
MOV.L   #function_address, r0   ; r0 = function to call

; Assembler produces
MOV.L   @(n, PC), r0            ; r0 = function to call

... around n+4 bytes later ...
.data.l function_address        ; constant stored in code segment
```

The notation used by the Microsoft SH-3 assembler is that the name of a label is treated as its address. You don't need to say `offset` like you do in the Microsoft 80386 assembler.

We also prepare the parameters for the call. As we noted when we discussed the calling convention, the first four parameters go in registers *r4* through *r7*, and the rest go on the stack.

In practice, the parameters will be prepared in whatever order the compiler finds convenient, and they will be interleaved with the code that prepares the function address (and with each other) in order to improve scheduling.

The final instruction for setting up the parameters can go into the branch delay slot, provided it does not use a PC-relative addressing mode.

```
    MOV.L   #function, r0           ; r0 = function to call
    MOV.L   @(24, r15), r5          ; r5 = local variable
    JSR     @r0                     ; call the function
    MOV.L   #large_constant, r4     ; r4 = some large constant
    ^^^^^ ILLSLOT EXCEPTION         ; (in the branch delay slot)
```

The `MOV.L #large_constant, r4` will be encoded by the assembler as a PC-relative load, which is illegal in a branch delay slot. Fortunately, the assembler will not let you do this:

```
error A151: Can't compute PC displacement in a delay slot
```

To fix this, you'll have to move the PC-relative load out of the delay slot, preferably by swapping it with some instruction that it is not dependent upon.

```
    MOV.L   #function, r0           ; r0 = function to call
    MOV.L   #large_constant, r4     ; r4 = some large constant
    JSR     @r0                     ; call the function
    MOV.L   @(24, r15), r5          ; r5 = local variable
                                    ; (in the branch delay slot)
```

Calling a function through a global variable function pointer (such as through the import address table, in the case of a function that was declared as `__declspec(import)`) involves two memory accesses, one to get the address of the global variable, and another to get the code pointer.

```
    MOV.L   #variable, r0            ; r0 = variable that holds the fptr
    MOV.L   @r0, r0                 ; r0 = the address to call
    JSR     @r0                     ; call it
```

Here and in the subsequent examples, I've removed the parameter-loading instructions.

Calling a virtual function means getting the function address from the object's vtable.

```
    MOV     r8, r4                  ; r4 = "this" for function call
    MOV.L   @r4, r0                 ; load vtable pointer into r0
    MOV.L   @(n, r0), r0            ; load function pointer from vtable into r0
    JSR     @r0                     ; call it
```

And calling a naïvely-imported function means calling a stub.

```
    MOV.L   #stub_address, r0        ; r0 = pointer to stub function
    JSR     @r0                      ; call it

    ...
stub:
    MOV.L   #__imp__Function, r0     ; r0 = pointer to IAT entry
    MOV.L   @r0, r0                  ; r0 = the address to call
    JMP     @r0                      ; and jump there
    NOP                              ; (branch delay slot)
    .data.l __imp__Function          ; address of IAT entry
                                     ; (constant for first MOV.L instruction)
```

Our last common pattern for today is the dense switch statement.

```
    switch (value) {
    case 1: ...
    case 2: ...
    case 3: ...
    case 4: ...
    case 5: ...
    default: ...
    }

        ADD     #-1,r4               ; bias by lowest valid value
        MOV     #4,r3                ; is it in the range of our jump table?
        CMP/HI  r3,r4
        BT      default              ; N: go to default case
        MOV.L   #jump_table, r2      ; get address of jump table
        MOV     r4,r0                ; prepare for indexed addressing
        MOV.B   @(r0,r2),r0          ; r0 = instruction offset for case
        NOP                          ; (we'll see more about this nop later)
        BRAF    r0                   ; jump to appropriate handler
        NOP                          ; (nothing in the branch delay slot)

    ...
jump_table:
        .data.b 0x0
        .data.b 0x1a
        .data.b 0x2c
        .data.b 0x42
        .data.b 0x78
```

The code first subtracts the lowest non-default case value, producing an index so that all the interesting cases are in the range 0 to *n* for some *n*. If the value is not in that range, then we jump to the `default:` . Otherwise, we use the index as an index into a jump table of bytes, and use a `BRAF` instruction to perform a relative jump.

If there is a case label more than 127 bytes away from the `BRAF` , then the jump table expands to contain word offsets, and the index needs to be doubled before being looked up.

```
ADD     #-1,r4              ; bias by lowest valid value
MOV     #4,r3               ; is it in the range of our jump table?
CMP/HI  r3,r4
BT      default             ; N: go to default case
MOV.L   #jump_table, r2     ; get address of jump table
MOV     r4,r0               ; prepare for indexed addressing
ADD     r0,r0               ; convert byte offset to word offset
MOV.W   @(r0,r2),r0         ; r0 = instruction offset for case
BRAF    r0                  ; jump to appropriate handler
NOP                         ; (nothing in the branch delay slot)
```

We double the index by adding it to itself (`add r0, r0`). This is where the extra `NOP` from the previous case comes into play. The compiler leaves a `NOP` in its code generation so it can choose the size of the jump table later without having to go back and recalculate all its offsets.

In theory the compiler could have emitted the jump table directly into the code rather than dropping just the address of the jump table, which then needs to be indirected through in order to access the actual jump table. That has its drawbacks though: You have a potentially large jump table in your code, which pushes the jump targets further away and makes it more likely you're going to need a bigger table. And having the possibility of a variable-sized table means that the calculation of jump offsets requires multiple passes until all the consequences have stabilized. It's easier for the compiler to just generate a pointer to a jump table and figure out the jump table later.

I guess in theory if there is more than 64KB of code in the `switch` statement, the jump table might have to contain longword offsets, and the `NOP` becomes a `SLL2` to scale the index up so it can access a longword array. I've never seen a function so large that this became an issue, though.

Next time, we'll wrap up this whirlwind tour of the SH-3 processor by walking through some actual code.

Raymond Chen

Follow