The SuperH-3, part 12: Calling convention and function prologues/epilogues

devblogs.microsoft.com/oldnewthing/20190820-00

August 20, 2019

Raymond Chen

The calling convention used by Windows CE for the SH-3 processor looks very much like the calling convention for other RISC architectures on Windows.

The short version is that the first four parameters (assuming they are all 32-bit integers) are passed in registers *r4* through *r7*, and the rest go onto the stack after a 16-byte gap. The 16-byte gap is the home space for the register parameters, and even if a function accepts fewer than four parameters, you must still provide a full 16 bytes of home space.

More strictly, the first 16 bytes of parameters are passed in registers *r4* through *r7*. If a parameter is a floating point type, then how it gets passed depends on how the parameter is declared in the function prototype.

- If the floating point type is prototyped as non-variadic, then it goes into the corresponding register *fr4* through *fr7*, and the integer register goes unused.
- If the floating point type is prototyped as variadic, then it stays in the integer register.
- If the function has no prototype, then the floating point type goes into both the floating point register and the integer register.

The reason for this rule is the same as before. Variadic parameters go into integer registers because the callee doesn't know what type they are upon function entry. To make things easier, variadic parameters are always passed in integer registers, so that the callee can just spill them into the home space and treat them all as stack-based parameters. And unprototyped functions pass the floating point values in both floating point and integer registers because it doesn't know whether the function is going to treat them as variadic or non-variadic, so it has to cover both bases.

Unlike the Windows calling convention for the MIPS R4000, the Windows calling convention for the SH-3 does not require 64-bit values to be 8-byte aligned. For example:

```
void f(int a, __int64 b, int c);
```

MIPS	Contents	SH-3	Contents
a0	*a*	*r4*	*a*
a1	unused	*r5*	*b*
a2	*b*	*r6*	
a3		*r7*	*c*
on stack	*c*		

On entry to the function, the return address is provided in the *pr* register, and on exit the function's return value is placed in the *r0* register. However, if the function's return value is larger than 32 bits, then a secret first parameter is passed which is a pointer to a buffer to receive the return value. The parameters are caller-clean; the function must return with the stack pointer at the same value it had when control entered.

If the concept of home space offends you, you can think of it as a 16-byte red zone that sits above the stack pointer.

The stack for a typical function looks like this:

⋮	
param 6	(if function accepts more than 4 parameters)
param 5	(if function accepts more than 4 parameters)
param 4 home space	
param 3 home space	
param 2 home space	
param 1 home space	← stack pointer at function entry
saved registers ⋮	
saved return address	← stack pointer after saving registers
local variables ⋮	
outbound parameters beyond 4 (if any) ⋮	

param 4 home space
param 3 home space
param 2 home space
param 1 home space ← stack pointer after prologue complete

The function typically starts by pushing onto the stack any nonvolatile registers, as well as its return address. This takes advantage of the pre-decrement addressing mode. In practice, the Microsoft C compiler allocates nonvolatile registers starting at *r8* and increasing, and preserves them on the stack in that order, followed by the return address.

In this example, the function has four registers to save, plus the return address.

```
function_start:
    MOV.L   r8, @-r15   ; push r8
    MOV.L   r9, @-r15   ; push r9
    MOV.L   r10, @-r15  ; push r10
    MOV.L   r11, @-r15  ; push r11
    STS.L   pr, @-r15   ; push pr
```

At some point (perhaps not immediately), the function will adjust its stack pointer to create space for its local variables and outbound parameters. If the function has a small stack frame, it can use the immediate form of the `SUB` instruction. Otherwise, it's probably going to load a constant into a register and use that as the input to the two-register form of the `SUB` instruction.

If the function has a large stack frame, it will be difficult to access variables far away from *r15* due to the limited reach of the *register indirect with displacement* addressing mode. To help with this problem, the compiler might park the frame pointer register *r14* in the middle of the frame, or at least close to a frequently-used variable, so that it can reach more local variables in a single instruction.

At the exit of the function, the operations performed in the prologue are reversed: The stack pointer is adjusted to point to the saved return address, and the saved registers are popped off the stack. Finally, the function returns with a `rts`.

```
    LDS.L   @r15+, pr   ; pop pr
    MOV.L   @r15+, r11  ; pop r11
    MOV.L   @r15+, r10  ; pop r10
    MOV.L   @r15+, r9   ; pop r9
    RTS                 ; return
    MOV.L   @r15+, r8   ; pop r8 (in the delay slot)
```

Lightweight leaf functions are those which call no other functions and which can accomplish their task using only volatile registers and the 16 bytes of home space. Such functions may not modify the *pr* register or any nonvolatile registers (which includes the stack pointer).

Next time, we'll look at some code patterns you'll see in the compiler-generated code, y'know, the stuff that goes *inside* the function. We'll start with misaligned data.

Raymond Chen

Follow