The MIPS R4000, part 13: Function prologues and epilogues

devblogs.microsoft.com/oldnewthing/20180418-00

April 18, 2018

Raymond Chen

We saw last time how functions are called. Today we'll look at the receiving end of a function call.

As noted earlier, all functions (except for lightweight leaf functions) must declare unwind codes in the module metadata so that the kernel can figure out what to do if an exception occcurs.

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
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

On entry to the function, the first four parameters are in registers, but they have reserved home space on the stack. Even if a function has fewer than four parameters, there is home space for all four registers. If there are more than four parameters, then those beyond the fourth are on the stack.

The function prologue needs to move the stack pointer down to make room for the local stack frame. The *local variables* include the return address and any saved registers. After the local variables come the outbound parameters (either directly on the stack for parameters beyond 4, or home space for the four register-based parameters). Again, even if a function accepts fewer than four parameters, it gets a full four words of home space.[1]

The 1992 compiler organized the local variables with the declared function local variables at higher addresses, followed by saved registers, and the return address closest to the outbound parameters. By 1995, the compiler started exploring other ways of organizing its local variables.

A typical function prologue looks like this:

```
ADDIU   sp, sp, -n1 ; carve out a stack frame
SW      ra, n2(sp)  ; save return address
SW      s1, n3(sp)  ; save nonvolatile register
SW      s0, n4(sp)  ; save nonvolatile register
```

The prologue must start by updating the stack pointer, and then it can store its registers in any order. You are allowed to interleave instructions from the function body proper into the prologue, provided they are purely computational instructions (no branches or memory access), and provided they do not mutate `sp`, `ra`, or any nonvolatile registers.[2] In practice, the Microsoft compiler does not take advantage of this.

To return from a function, the function places the return value, if any, in the *v0* register and possibly the *v1* register. It then executes the formal function epilogue:

```
MOVE    v0, return_value
LW      s0, n4(sp)  ; restore nonvolatile register
LW      s1, n3(sp)  ; restore nonvolatile register
LW      ra, n2(sp)  ; restore return address
JR      ra          ; return to caller
ADDIU   sp, sp, n1  ; restore stack pointer (in branch delay slot)
```

Notice that the adjustment of the stack pointer happens as the very last thing, even after the return instruction! That's because it sits in the branch delay slot, so it executes even though the branch is taken.

[1] If a function uses `alloca`, then the memory is carved out between the existing local variables and the outbound parameters.

[2] This rule exists so that when the exception unwinder needs to reverse-execute a function prologue, it can just ignore the instructions it doesn't understand.

Raymond Chen

Follow