The PowerPC 600 series, part 10: Passing parameters, function prologues and epilogues

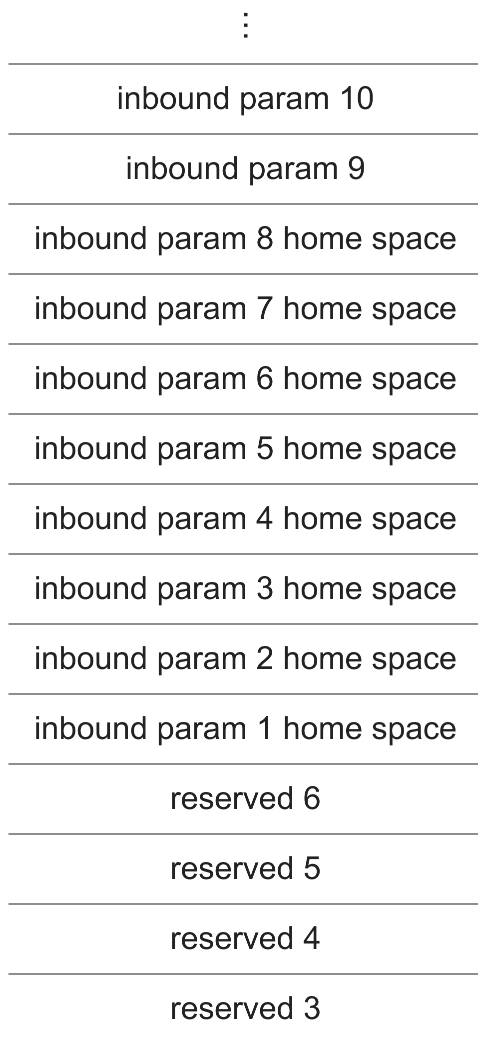
devblogs.microsoft.com/oldnewthing/20180817-00

August 17, 2018

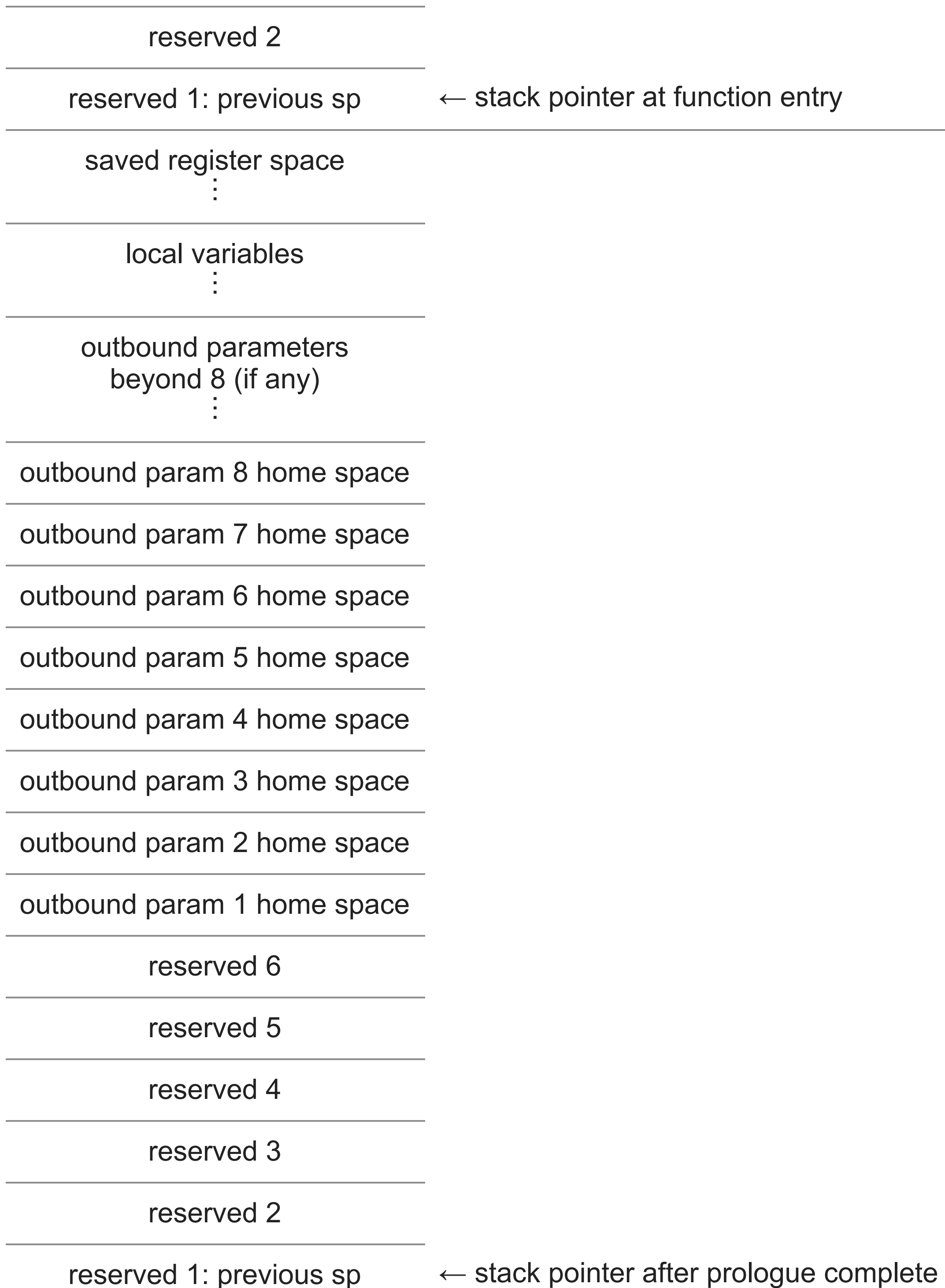
Raymond Chen

We saw a little bit of the Windows NT software convention with our introduction to the table of contents. Today we'll start looking at the conventions related to the stack. (Believe it or not, this will connect back to the table of contents, but it'll take a while before we get there.)

The format of the stack is as follows:



⋮
inbound param 10
inbound param 9
inbound param 8 home space
inbound param 7 home space
inbound param 6 home space
inbound param 5 home space
inbound param 4 home space
inbound param 3 home space
inbound param 2 home space
inbound param 1 home space
reserved 6
reserved 5
reserved 4
reserved 3



That’s a big stack.

Starting at the top of the diagram (deepest on the stack) are the stack-based parameters, which are the parameters beyond the first 8.

Next is home space for the first 8 parameters. Those parameters are passed in registers, but reserve space for them on the stack in case the function needs to spill them. Even if the function has fewer than eight parameters, there is home space for all eight of them.

Integer parameters are passed in *r3* through *r10*, and floating point parameters come in *f1* through *f13*. The register assignment is like the Alpha AXP and MIPS, where each parameter can go into either an integer or floating point register, and if you use one, then the other goes unused.

After the home space come 24 bytes (6 words) of system-reserved space. One of them is required to hold the previous stack pointer, as we'll see soon. The others are uninitialized.

At entry to the function, the prologue needs to set up its own stack frame. It saves nonvolatile registers into the saved variable space and then atomically updates the stack pointer and links it to the previous stack frame. Here's a sample prologue. I've added blank lines to separate the major sections.

```
01ae2398 7c0802a6  mflr    r0           ; move return address to r0

01ae239c 93c1fff8  stw     r30,-8(r1)   ; save nonvolatile register
01ae23a0 93e1fffc  stw     r31,-4(r1)   ; save nonvolatile register

01ae23a4 9001fff4  stw     r0,-0xC(r1)  ; save return address

01ae23a8 9421ffb0  stwu    r1,-0x50(r1) ; create stack frame
```

The first thing a function does is save the link register in *r0* so it doesn't lose the return address. In my experience, the only thing the Microsoft compiler uses the *r0* register for is transferring to and from the link register.

The next thing a function does is save to the stack the nonvolatile registers it intends to use. (Recall that *r1* is the stack pointer register.) This function uses two nonvolatile registers *r30* and *r31*, and it saves them onto the stack immediately below the stack pointer, in order. I'm not sure if it's a requirement of the software convention, but the Microsoft compiler always allocates its nonvolatile registers top-down, so that the set of nonvolatile registers is a contiguous range ending at *r31*.[1] Furthermore, it always saves the registers in the same place: *r31* goes on the stack first, then *r30*, and so on. Even if it's not a requirement, the Microsoft compiler is pretty consistent about it, which makes unwinding the stack in the debugger a lot easier because you always know that, for example, the saved value of *r29* is at offset −12 from the inbound *sp*.[2]

The third step is saving the *r0* register (which holds the return address). The Microsoft compiler always stores the return address immediately below the saved registers. Again, I don't know if it's a requirement, but it's a handy thing to take advantage if you need to manually unwind the stack.

The final step of the prologue is creating the stack frame with the `stwu` instruction. This instruction stores the current stack pointer at the specified negative offset from the top of the stack (creating the next node in the linked list) and then updates the stack pointer to the

address it just stored to. This all happens atomically in a single instruction, which means that the linked list of stack frames is always preserved at any moment in time. This is great for sampling profilers, which might otherwise have a hard time building a proper stack trace if it happened to catch the prologue at a bad time.

The compiler is permitted to advance instructions from the function body proper into the prologue, provided it doesn't alter any nonvolatile registers or perform any branches.

The function epilogue also follows a consistent pattern:

```
01ae2444 7ca32b78  mr      r3,r5        ; set return value

01ae2448 80010044  lwz     r0,0x44(r1)  ; load return address

01ae244c 83c10048  lwz     r30,0x48(r1) ; restore nonvolatile register
01ae2450 83e1004c  lwz     r31,0x4C(r1) ; restore nonvolatile register

01ae2454 7c0803a6  mtlr    r0           ; move return address to link register

01ae2458 38210050  addi    r1,r1,0x50   ; pop the stack frame

01ae245c 4e800020  blr                  ; return
```

The main body of the function ends with the desired return value in *r3*. At this point, we enter the epilogue.

First, the epilogue loads the return address from the stack. These offsets are different from the ones used at the start of the function because they were saved before the frame was pushed, but they are being restored whlie the frame is still active. Since the size of the stack frame is 80 bytes, the values will differ by 80.

Next, the epilogue restores the nonvolatile registers.

Step three is moving the return address into the link register in preparation for the actual return.

Step four is popping off the stack frame by moving the stack pointer back to where it was when the function started.

The last step is to return back to the caller with the Windows NT-approved `blr` instruction.

Function prologues and epilogues are tightly-controlled because the system exception dispatcher needs to be able to unwind a function's stack even when it's in the middle of a prologue or epilogue. This means that the system needs to be able to reverse-execute a prologue and forward-execute an epilogue in order to get the registers properly set up when

dispatching an exception to the caller of the function that was interrupted. (The function that was interrupted cannot have an exception handler in place because exception handlers cannot be active during a prologue or epilogue.)

The fact that the initial portion of the stack frame is constructed at negative offsets from the stack pointer means that the system must have a large enough red zone to accommodate the worst-case scenario of a function that needs to save all of the nonvolatile registers, plus the return address.

So let's do some math. Integer registers *r14* through *r31* are nonvolatile, so that's 18 × 4 = 72 bytes for nonvolatile integer registers. Floating point registers *f14* through *f31* are also nonvolatile, and floating point registers are 8 bytes in size, so that means another 18 × 8 = 144 bytes, added to the 72 we already have makes 216. And then there are the stragglers:

- Parts of the condition register are also nonvolatile, and in practice you just save the whole thing,
- Similarly, the floating point control register.
- The return address.

That adds twelve more bytes, bringing us to 232 bytes. Since the stack must be 8-byte aligned, we round up to the next multiple of 8, but hey, it's already a multiple of 8, so we're good. [Corrected from 16.]

Exercise: Why don't we need to count the system reserved bytes (specifically the the link to the previous stack frame) toward the red zone?

At the start of this entry, I promised that this would lead to the table of contents eventually. We're almost there. The story continues next time.

Bonus chatter: I lied when I said that the prologue cannot contain any branch instructions. There is one branch instruction that is specifically permitted: A call to a helper function to spill the registers. There could be a lot of registers to spill, and the software convention permits you to use helpers function for the following operations:

- Bulk-saving integer registers.
- Bulk-saving floating point registers.
- Bulk-restoring integer registers.
- Bulk-restoring floating point registers.

These bulk save/restore functions must follow a specific format so that the exception unwinder understands how to recover in case an interrupt occurs inside the helper. The details are not important aside from knowing that they use the *r12* register to specify where the registers go. (Obviously they can't use the standard calling convention because those registers are being used by the function whose prologue is being executed!)

Bonus bonus chatter: The size of the red zone is described in the `ntppc.h` header file as

```
#define STK_SLACK_SPACE 232
```

It didn't explain how the number 232 was arrived at.

The x64 software conventions for Windows NT are well-documented, but I couldn't find any documents covering the older platforms. All of the software conventions for the PowerPC were reverse-engineered by studying compiler output and reading very old kernel source code.[3]

[1] Doing it this way allows the bulk save/restore functions to be shared among multiple functions. Special "store multiple contiguous registers" and "load multiple contiguous registers" instructions are available in big-endian mode, but not in little-endian mode. In little-endian mode, you have to save them one at a time, hence the bulk save/restore helpers.

[2] Well, not always. If floating point registers need to be saved, they get saved first. But you don't see floating point in system code much, so in practice you can usually get away with pretending they don't exist.

[3] The code has some nice diagrams in the comments about the stack layout. Too bad those diagrams are wrong. I suspect the ABI was redesigned at some point, and the comments and diagrams weren't fully updated to match.

Raymond Chen

Follow