# The MIPS R4000, part 12: Calling convention

devblogs.microsoft.com/oldnewthing/20180417-00

April 17, 2018

Raymond Chen

The Windows NT calling convention for the MIPS R4000 is similar to the other major MIPS calling conventions, but calling conventions for the MIPS are like snowflakes: Despite being made of the same underlying materials, no two are completely alike.

The short version of the parameter passing is that the first four parameters are passed in registers *a0* through *a3*, and the remaining parameters go on the stack after a 16-byte gap. The 16-byte gap represents the home space for the register-based parameters. We've seen this convention before, in the x64 calling convention. Even if a function accepts fewer than four parameters, you must still provide a full 16 bytes of home space.

Things get weird when you mix in 64-bit values or floating point. The way to think about it is as if you were creating a C structure whose members are all the parameters, in order, except that any types smaller than a 32-bit value are promoted to a 32-bit value. If you have a 64-bit value (either integer or floating point), you may need to insert padding to get the parameter to be properly aligned.

Once you've laid out your parameters in the structure, you load the first sixteen bytes into *a0* through *a3*, and the rest go on the stack. However, if a parameter that would normally be passed in *a0* through *a3* turns out to be a non-variadic floating point value, then it is stored in *$f12*/*$f13* (for the first floating point value) or *$f14*/*$f15* (for the second), and the corresponding integer register is left unused.

Here are some examples:

```
void f(int a, char b, short c, int d, int e);
```

| Offset | Parameter | Passed as |
|---|---|---|
| `00` | `int a` | *a0* |
| `04` | `int b` | *a1* |
| `08` | `int c` | *a2* |

| | | |
|---|---|---|
| `0C` | `int d` | *a3* |
| `10` | `int e` | `0x10(sp)` |

```
void f(float a, int b, double c, int d);
```

| Offset | Parameter | Passed as |
|---|---|---|
| `00` | `float a` | *f12* |
| `04` | `int b` | *a1* |
| `08` | `double c` | *f14* |
| `0C` | | *f15* |
| `10` | `int e` | `0x10(sp)` |

```
void f(int a, double b, float c);
```

| Offset | Parameter | Passed as |
|---|---|---|
| `00` | `int a` | *a0* |
| `04` | padding | |
| `08` | `double b` | *f12* |
| `0C` | | *f13* |
| `10` | `float c` | `0x10(sp)` |

```
void f(int a, ...);
f(1, 2, 0.0, 3);
```

| Offset | Parameter | Passed as |
|---|---|---|
| `00` | `1` | *a0* |
| `04` | `2` | *a1* |
| `08` | `0.0` | *a2* |
| `0C` | | *a3* |
| `10` | `3` | `0x10(sp)` |

In this last example, the floating point double-precision value `0.0` is a variadic parameter (matches the `...` part of a function prototype), so it gets passed in the integer registers even though it's a floating point value. That's because one of the first things that variadic functions do is spill all their variadic register parameters onto the stack so they form a contiguous array of bytes. Passing all variadic parameters in integer registers means that this spilling can be done without knowing the types of the parameters. (Which is a good thing because the types of the parameters are frequently not known at compile time.)

The last wrinkle is if you're calling a function with no prototype. In that case, you don't know whether a parameter is variadic or not. If the parameter is a floating point value, then you have to pass it in both an integer register *and* a floating point register, because you don't know where the callee is going to look for it.

```
f(1, 2, 0.0, 3); // no prototype
```

| Offset | Parameter | Passed as |
|---|---|---|
| `00` | `1` | *a0* |
| `04` | `2` | *a1* |
| `08` | `0.0` | *a2* and *f12* |
| `0C` | | *a3* and *f13* |
| `10` | `3` | stack |

This explains the importance of the rule that if a parameter is passed in a floating point register, then the corresponding integer register is left unused. Without that rule, calling functions with no prototype would be a disaster because the register assignment would be different depending on whether the function takes variadic parameters or not.

With the exception of lightweight leaf functions, every function must include exception unwind codes in the module metadata so that the kernel can figure out what to do if an exception occurs.

A lightweight leaf function is one that can do its work using only the 16 bytes of home space, plus any scratch registers. It cannot move the stack pointer or modify any callee-preserved registers. Furthermore, the return address must remain in the *ra* register for the duration of the function.

You are allowed to promote your lightweight leaf function to a full function by a technique known as *shrink-wrapping*, which I described earlier.

(Some of the details of the calling convention are documented on MSDN. The documentation was originally written for Windows CE, but I figure they are still true for Windows NT, because why not reuse the compiler you already have?)

Raymond Chen

**Follow**