The MIPS R4000, part 1: Introduction

devblogs.microsoft.com/oldnewthing/20180402-00

April 2, 2018

Raymond Chen

Continuing in the "Raymond introduces you to a CPU architecture that Windows once supported but no longer does" sort-of series, here we go with the MIPS R4000.

The MIPS R4000 implements the MIPS III architecture. It is a 64-bit processor, but Windows NT used it in 32-bit mode. I'll be focusing on the aspects of the processor relevant to debugging user-mode programs on Windows NT. This means that I may skip over various technical details on the assumption that the compiler knows what the rules are and won't (intentionally) generate code that violates them.

Throughout, I will say "MIPS" instead of "MIPS III architecture". Some of the issues do not apply to later versions of the architecture family, but I am focusing on MIPS III since that's what Windows NT used.

The MIPS is a RISC-style load-store processor: The only operations you can perform with memory are load and store. There is no "add value to memory" instruction, for example. Each instruction is 32 bits wide, and the program counter must be on an exact multiple of 4.

The processor can operate in either little-endian or big-endian mode; Windows NT uses little-endian mode, and even though some instructions change behavior depending on whether the processor is in big-endian or little-endian mode, I will discuss only the little-endian case.

The architectural terminology for a 32-bit value is a *word* (w), and a 16-bit value is a *halfword* (h). There's also *doubleword* (d) for 64-bit values, but we won't see it here because we are focusing on the 32-bit mode of the processor.

The MIPS has 32 general-purpose integer registers, formally known as registers *$0* through *$31*, but which conventionally go by these names:

Register	Mnemonic	Meaning	Preserved?	Notes
$0	*zero*	reads as zero	Immutable	Writes are ignored

$1	*at*	assembler temporary	Volatile	Helper for synthesized instructions
$2	*v0*	value	No	On function exit, contains the return value
$3	*v1*	value	No	High 32 bits of return value (for 64-bit values)
$4…$7	*a0…a3*	argument	No	On function entry, contains function parameters
$8…$15	*t0…t7*	temporary	No	
$16…$23	*s0…s7*	saved	Yes	
$24…$25	*t8…t9*	temporary	No	
$26…$27	*k0…k1*	kernel	No access	Reserved for kernel use
$28	*gp*	global pointer	Yes	Not used by 32-bit code
$29	*sp*	stack pointer	Yes	
$30	*s8*	frame pointer	Yes	For functions with variable-sized stacks
$31	*ra*	return address	Maybe	

The *zero* register reads as zero, and writes to it are ignored.

The *k0* and *k1* registers are reserved for kernel use, and no well-written user-mode program will use them.[1]

Win32 requires that the *sp* and *s8* registers be used for their stated purpose throughout the entire function. If a function does not have a variable-sized stack frame, then it can use *s8* for any purpose (which is why the disassembler calls it *s8* instead of *fp*, I guess). And since 32-bit code doesn't ascribe special meaning to *gp*, then it too can be used for any purpose, provided its value is preserved across the call. In practice the Microsoft compiler merely avoids the *gp* register completely, and it uses the *s8* register only as a frame pointer.

The stack is always aligned on an 8-byte boundary, and there is no red zone.

Some registers have stated purposes only at entry to a function or exit from a function. When not at the function boundary, those registers may be used for any purpose.

Register marked with "Yes" in the "Preserved" column must be preserved across the call; those marked "No" do not.

The *ra* register is marked "Maybe" because you don't normally need to preserve it. However, if you are a leaf function that does not modify any preserved registers (not even *sp*), then you can skip the generation of unwind codes for the leaf function, but you must keep the return address in *ra* for the duration of your function so that the operating system can unwind out of the function should an exception occur. (Special rules for lightweight leaf functions also exist for Itanium, Alpha AXP, and x64.)

The *at* register is volatile because the assembler can use it for various invisible purposes, primarily for synthesizing missing instructions. We'll see examples of this as we go.

There are also two special-purpose integer registers, called *HI* and *LO*. These are used by multiplication and division instructions, and we'll cover them when we get to multiplication and division.

There are 32 single-precision (32-bit) floating point registers, which can be paired up to form 16 double-precision (64-bit) floating point registers. When a pair is used to operate on a single-precision value, the lower-numbered register holds the value, and the higher-numbered register is not used. (Indeed, the value in the higher-numbered register will be garbage.) So I guess you really have just 16 single-precision floating point registers, since the odd-numbered ones are basically useless.

Register(s)	Meaning	Preserved?	Notes
$f0*/*$f1	return value	No	
$f2*/*$f3	second return value	No	For imaginary component of complex number.
$f4*/*$f5…*$f10*/*$f11*	temporary	No	
$f12*/*$f13…*$f14*/*$f15*	arguments	No	
$f16*/*$f17…*$f18*/*$f19*	temporary	No	
$f20*/*$f21…*$f30*/*$f31*	saved	Yes	

Floating point support is optional. If not supported, floating point instructions will trap into the kernel, and the kernel is expected to emulate the instruction.

There is not a lot of floating point in typical systems programming, so I won't cover it except when discussing the calling convention later.

There is no flags register. Hopefully you don't find this weird any more, seeing as we already encountered this with the Alpha AXP.

The 32-bit address space is split down the middle between user-mode and kernel-mode. The kernel-mode space is further split: Half of the kernel-mode address space is dedicated to mapping physical addresses (the lowest 512MB[2] gets mapped twice, once cached and once uncached), leaving only 1GB for the operating system. This partitioning is architectural; you don't get a choice in the matter.

Okay, we'll begin next time by looking at 32-bit integer calculations.

[1] I know you're wondering what happens if poorly-written user-mode code tries to use them. The answer is that user-mode code can modify the register all it wants, but the value read back may not be equal to value last written. As far as user mode is concerned, it's basically a black hole register that reads as garbage. This makes it even more useless than the *zero* register, which is a black hole register that at least reads as zero. (Internally, the registers are used by kernel mode as scratch variables during interrupt and exception handling.)

[2] I guess they figured that if you had more than 512MB of RAM, you'd have switched to a 64-bit operating system.

Raymond Chen

Follow