The PowerPC 600 series, part 1: Introduction

devblogs.microsoft.com/oldnewthing/20180806-00

August 6, 2018

Raymond Chen

The PowerPC is a RISC processor architecture which grew out of IBM's POWER architecture. Windows NT support was introduced in Windows NT 3.51, and it didn't last long; the last version to support it was Windows NT 4.0. Despite not being supported by the flagship operating system, it continued to be supported by Windows CE, and a later version of the PowerPC was chosen as the processor for the Xbox 360.

As with all the processor retrospective series, I'm going to focus on how Windows NT used the PowerPC in user mode because the original audience for all of these discussions was user-mode developers trying to get up to speed debugging their programs on PowerPC.

The PowerPC 600 series started out as a 32-bit processor, with 64-bit support arriving in the 620. The earliest record I can find (not that I looked very hard) shows Windows NT supporting the 603 and 604 processors. I guess this makes sense, because Wikipedia says that the 603 was the first processor to support the full PowerPC instruction set. The 603 could complete a maximum of two instructions per cycle; the 604 could do up to four. The 603 did not have a dynamic branch predictor, but the 604 did. Both could forward arithmetic operations into the next arithmetic operation, so consecutive integer arithmetic operations did not stall, even if the second depended on the result of the first.

The PowerPC 600 series processors are natively big-endian, with an option for little-endian operation. Windows NT uses the processor in 32-bit little-endian mode.[1] Even though the processor can be put into little-endian mode, this affects only how bytes are swapped when they are read from or written to memory; the instructions themselves still operate in a big-endian way, Among other things, the bits in a register are numbered from most-significant to least-significant: Bit 0 is the high-order bit, and bit 31 is the low-order bit.

The PowerPC has 32 integer registers, each 32 bits wide. They are officially named *GPR0* through *GPR31*, but the assembler just calls them *0* through *31*. This is ridiculously confusing,[2] so nobody uses the purely numeric names. People call them *r0* through *r31*. (Some assemblers call them *r.0* through *r.31*.)

Register	Mnemonic	Meaning	Preserved?	Notes
gpr0	*r0*		No	Of limited use
gpr1	*r1*	stack pointer	Yes	Includes 232-byte negative red zone
gpr2	*r2*	table of contents	Yes, mostly	Access to global variables
gpr3…gpr10	*r3…r10*	argument	No	On function entry, contains function parameters
gpr11	*r11*	temporary	No	For function glue
gpr12	*r12*	temporary	No	prologue and epilogue helper
gpr13	r13	read-only	Yes	TEB
gpr14…gpr31	*r14…r31*	saved	Yes	

Note that this does not exactly line up with the PowerPC register conventions for other platforms. (Many other platforms assign special meanings to *gpr11* through *gpr13*.)

The stack must be kept on an 8-byte boundary. There is a large red zone of 232 bytes at negative offsets from the stack pointer. We'll see the importance of this when we look at function prologues.

The function return value is placed in *r3*.

The *r0* register is of limited use because many instructions cannot use a source of *r0*. We'll see more about that later.

We'll learn about the table of contents, function glue, and epilogue/prologue helpers later when we cover Windows NT software conventions.

In addition to the general-purpose integer registers, there are a number of special-purpose 32-bit integer registers. There are only nineteen of these special-purpose registers, but the numbers range from *spr1* to *spr1013*. (The number space is very sparsely populated, but I guess they reserved room for adding more registers in the future.) These are the ones you're likely to see in user-mode code:

Register	Mnemonic	Meaning	Preserved?	Notes
spr1	*xer*	Status bits	No	Integer exception register

spr8	*lr*	link register	No	On function entry, contains return address
spr9	*ctr*	counter	No	Dedicated counter or jump target
fpscr	*fpscr*	Status bits	?	Floating point status and control register

I've never had to deal with floating point on the PowerPC, so I don't know what parts of *fpscr* need to be preserved and what parts don't.

We'll learn more about the other special registers as the need arises.

Remember how the Itanium, MIPS, and Alpha don't have a flags register? Well, the PowerPC scoffs at them. "Flags register? You say you want a flags register? I've got your flags register right here. In fact, I've got *eight sets* of flags registers." They are named *cr0* through *cr7*, each four bits wide. (The "cr" stands for *condition register*.) The pseudo-register *cr* can be used to treat them as one giant 32-bit register.[3] Remember that the PowerPC is a big-endian processor, so *cr0* occupies the most significant bits of *cr*, and so *cr7* occupies the least significant bits.

Condition register *cr0* is the implicit target of integer operations, and condition register *cr1* is the implicit target of floating point operations. I don't know which condition registers must be preserved across calls, because I've never found any code that needed to.

The PowerPC also has 32 floating-point double-precision registers, officially named *FPR0* through *FPR31*.

Register	**Mnemonic**	**Preserved?**	**Notes**
fpr0	*f0*	No	temporary
fpr1…fpr13	*f1…f13*	No	Function parameters
fpr14…fpr31	*f14…f31*	Yes	

As for instruction encoding, each instruction is 32 bits wide and must be aligned on a four-byte boundary. The instruction whose encoding is `0x00000000` is reserved as an invalid instruction, so trying to execute a page of zeros will instantly fault.

The general syntax for multi-operand opcodes is

```
    opcode  destination, source1, source2, source3...
```

with the notable exception of store instructions, which put the source register on the left and the address destination on the right.

The architectural terms for operand sizes are *byte*, *halfword* (2 bytes), *word* (4 bytes), *doubleword* (8 bytes), and *quadword* (16 bytes). In 32-bit operation, the largest unit that can be operated on directly is the word.

In opcode names, the word *arithmetic* is used to emphasize that the operands are treated as signed (usually abbreviated `a`), and the words *logical* (`l`) and *unsigned* (`u`) or sometimes *zero-extended* (`z`) are used to emphasize that the operands are treated as unsigned. I guess they couldn't make up their mind what to call it unsigned operations, so they chose one at random each time they needed one. Note further that these conventions are not uniformly applied, so stay alert.

The processor maintains the fiction that every instruction is retired completely before the next one starts. Consequently, there are no architectural branch delay slots or load delay slots. It also means that when an exception is raised, all instructions preceding the exception have run to completion, and no instructions after the exception will appear to have started.

Internally, the processor may perform operations out of order or in parallel or speculatively, and it may introduce stalls if your dependencies are too close together, but the processor does its best to hide this from the code being executed.

There are two notable exceptions to the principle of sequential operation:

- Floating point exceptions in imprecise mode can be delayed beyond the instruction that triggered the exception.
- Self-modifying code requires special instructions to evict the old instructions out of the I-cache.

Both reads and writes to memory can be reordered, and reads can be speculated. Storing a value may partly succeed before raising an exception. (For example, an unaligned store that crosses into an invalid page may write to the valid page and then take an exception on the invalid page.)

Okay, that's enough background. We'll pick up next time by taking a closer look at those condition registers.

[1] When the processor is in 32-bit mode, you can still execute 64-bit instructions. However, since Windows NT did not require a 64-bit capable version of the PowerPC processor, PowerPC programs for Windows NT had to perform runtime detection of 64-bit support and run either a 32-bit friendly version of the code or a 64-bit version of the code. In practice,

nobody did this. They just stuck to 32-bit code. (Even though you could use 64-bit instructions in 32-bit mode, the ABI preserves only the least-significant 32 bits of saved registers.)

[2] The designers of the PowerPC assembly language appear to be dedicated to making their instruction set as confusing as possible by making the assembly language be just barely more readable than machine code. For example, to say "Decrement the counter, and branch if the result is zero and the *eq* flag is set in *cr3*", they want you to write

```
    bc  2, 14, destination
```

Because obviously 2 means "decrement counter and branch if the result is zero and the specific flag is set", and naturally 14 means "the *eq* flag in *cr3*."

The Windows disassembler substitutes names for some (but not all) of these magic numbers at disassembly so you don't have to remember all the codes.

[3] You might think, "Who's to say which is the real register and which is the pseudo-register? You could equivalently think of *cr* as the real register, and the *cr#* registers as pseudo-registers!" Perhaps so, but the processor can execute operations on different *cr#* registers in parallel. If *cr* were the real register, then you would expect multiple operations on different *cr#* registers to be dependent on each other since they are all operating on *cr*.

Raymond Chen

Follow