The Itanium processor, part 1: Warming up

devblogs.microsoft.com/oldnewthing/20150727-00

July 27, 2015

Raymond Chen

The Itanium may not have been much of a commercial success, but it is interesting as a processor architecture because it is different from anything else commonly seen today. It's like learning a foreign language: It gives you an insight into how others view the world.

The next two weeks will be devoted to an introduction to the Itanium processor architecture, as employed by Win32. (Depending on the reaction to this series, I might also do a series on the Alpha AXP.)

I originally learned this information in order to be able to debug user-mode code as part of the massive port of several million lines of code from 32-bit to 64-bit Windows, so the focus will be on being able to read, understand, and debug user-mode code. I won't cover kernel-mode features since I never had to learn them.

Introduction

The Itanium is a 64-bit EPIC architecture. EPIC stands for Explicitly Parallel Instruction Computing, a design in which work is offloaded from the processor to the compiler. For example, the compiler decides which operations can be safely performed in parallel and which memory fetches can be productively speculated. This relieves the processor from having to make these decisions on the fly, thereby allowing it to focus on the real work of processing.

Registers overview

There are a lot of registers.

- 128 general-purpose integer registers *r0* through *r127*, each carrying 64 value bits and a trap bit. We'll learn more about the trap bit later.
- 128 floating point registers *f0* through *f127*.
- 64 predicate registers *p0* through *p63*.
- 8 branch registers *b0* through *b7*.
- An instruction pointer, which the Windows debugging engine for some reason calls *iip*. (The extra "i" is for "insane"?)

- 128 special-purpose registers, not all of which have been given meanings. These are called "application registers" (*ar*) for some reason. I will cover selected register as they arise during the discussion.
- Other miscellaneous registers we will not cover in this series.

Some of these registers are further subdivided into categories like *static*, *stacked*, and *rotating*.

Note that if you want to retrieve the value of a register with the Windows debugging engine, you need to prefix it with an at-sign. For example `? @r32` will print the contents of the *r32* register. If you omit the at-sign, then the debugger will look for a variable called *r32*.

A notational note: I am using the register names assigned by the Windows debugging engine. The formal names for the registers are *gr#* for integer registers, *fr#* for floating point registers, *pr#* for predicate registers, and *br#* for branch registers.

Static, stacked, and rotating registers

These terms describe how the registers participate in register renumbering.

Static registers are never renumbered.

Stacked registers are pushed onto a register stack when control transfers into a function, and they pop off the register stack when control transfers out. We'll see more about this when we study the calling convention.

Rotating registers can be cyclically renumbered during the execution of a function. They revert to being stacked when the function ends (and are then popped off the register stack). We'll see more about this when we study register rotation.

Integer registers

Of the 128 integer registers, registers *r0* through *r31* are static, and *r32* through *r127* are stacked (but they can be converted to rotating).

Of the static registers, Win32 assigns them the following mnemonics which correspond to their use in the Win32 calling convention.

Register	Mnemonic	Meaning
r0		Reads as zero (writes will fault)
r1	*gp*	Global pointer
r8…r11	*ret0…ret3*	Return values

r12	*sp*	Stack pointer
r13		TEB

Registers *r4* through *r7* are preserved across function calls. Well, okay, you should also preserve the stack pointer and the TEB if you know what's good for you, and there are special rules for *gp* which we will discuss later. The other static variables are scratch (may be modified by the function).

Register *r0* is a register that always contains the value zero. Writes to *r0* trigger a processor exception.

The *gp* register points to the current function's global variables. The Itanium has no absolute addressing mode. In order to access a global variable, you need to load it indirectly through a register, and the *gp* register points to the global variables associated with the current function. The *gp* register is kept up to date when code transfers between DLLs by means we'll discuss later. (This is sort of a throwback to the old days of `MAKEPROCINSTANCE`.)

Every integer register contains 64 value bits and one trap bit, known as not-a-thing, or *NaT*. The NaT bit is used by speculative execution to indicate that the register values are not valid. We learned a little about NaT some time ago; we'll discuss it further when we reach the topic of control speculation. The important thing to know about NaT right now is that if you take a register which is tagged as NaT and try to do arithmetic with it, then the NaT bit is set on the output register. Most other operations on registers tagged as NaT will raise an exception.

The NaT bit means that accessing an uninitialized variable can *crash*.

```
void bad_idea(int *p)
{
 int uninitialized;
 *p = uninitialized; // can crash here!
}
```

Since the variable *uninitialized* is uninitialized, the register assigned to it might happen to have the NaT bit set, left over from previous execution, at which point trying to save it into memory raises an exception.

You may have noticed that there are four return value registers, which means that you can return up to 32 bytes of data in registers.

Floating point registers

Register	Meaning

f0	Reads as 0.0 (writes will fault)
f1	Reads as 1.0 (writes will fault)

Registers *f0* through *f31* are static, and *f32* through *f127* are rotating.

By convention, registers *f0* through *f5* and *f16* through *f31* are preserved across calls. The others are scratch.

That's about all I'm going to say about floating point registers, since they aren't really where the Itanium architecture is exciting.

Predicate registers

Instead of a flags register, the Itanium records the state of previous comparison operations in dedicated registers known as *predicates*. Each comparison operation indicates which predicates should hold the comparison result, and future instructions can test the predicate.

Register	Meaning
p0	Reads as *true* (writes are ignored)

Predicate registers *p0* through *p15* are static, and *p16* through *p63* are rotating.

You can predicate almost any instruction, and the instruction will execute only if the predicate register is *true*. For example:

```
(p1) add ret0 = r32, r33
```

means, "If predicate *p1* is *true*, then set register *ret0* equal to the sum of *r32* and *r33*. If not, then do nothing." The thing inside the parentheses is called the *qualifying predicate* (abbreviated *qp*).

Instructions which execute unconditionally are internally represented as being conditional upon predicate register *p0*, since that register is always *true*.

Actually, I lied when I said that the instruction will execute only if the qualifying predicate is *true*. There is one class of instructions which execute regardless of the state of the qualifying predicate; more on that later.

The Win32 calling convention specifies that predicate registers *p0* through *p5* are preserved across calls, and *p6* through *p63* are scratch.

There is a special pseudo-register called *preds* by the Windows debugging engine which consists of the 64 predicate registers combined into a single 64-bit value. This pseudo-register is used when code needs to save and restore the state of the predicate registers.

Branch registers

The branch registers are used for indirect jump instructions. The only things you can do with branch registers are load them from an integer register, copy them to an integer register, and jump to them. In particular, you cannot load them directly from memory or do arithmetic on them. If you want to do any of those things, you need to do it with an integer register, then transfer it to a branch register.

The Win32 calling convention assigns the following meanings to the branch registers:

Register	Mnemonic	Meaning
b0	*rp*	Return address

The return address register is sometimes called *br*, but the disassembler calls it *rp*, so that's what we'll call it.

The return address register is set automatically by the processor when a `br.call` instruction is executed.

By convention, registers *b1* through *b5* are preserved across calls, while *b6* and *b7* are scratch. (Exercise: Is *b0* preserved across calls?)

Application registers

There are a large number of application registers, most of which are not useful to user-mode code. We'll introduce the interesting ones as they arise. I've already mentioned one of them already: bsp is the ia64's second stack pointer.

Break

Okay, this was a whirlwind tour of the Itanium register set. I bet your head hurts already, and we haven't even started coding yet!

In fact, we're not going to be coding for quite some time. Next time, we'll look at the instruction format.

Raymond Chen

Follow