The Itanium processor, part 2: Instruction encoding, templates, and stops

devblogs.microsoft.com/oldnewthing/20150728-00

July 28, 2015

Raymond Chen

Instructions on Itanium are grouped into chunks of three, known as *bundles*, and each of the three positions in a bundle is known as a *slot*. A bundle is 128 bits long (16 bytes) and always resides on a 16-byte boundary, so that the last digit of the address is always zero. The Windows debugging engine disassembler shows the three slots as if they were at offsets 0, 4, and 8 in the bundle, but in reality they are all crammed together into one bundle.

You cannot jump into the middle of a bundle.

Now, you can't just put any old instruction into any old slot. There are 32 bundle *templates*, and each has different rules about what types of instructions they can accept and the dependencies between the the slots. For example, the bundle template *MII* allows a memory access instruction in slot 0, an integer instruction in slot 1, and another integer instruction in slot 2.

(Math: Each slot is 41 bits wide, so 123 bits are required to encode the slots. Add five bits for encoding the template, and you get 128 bits for the entire bundle.)[1]

The slot types are

- M = memory or move
- I = complex integer or multimedia
- A = simple arithmetic, bit logic, or multimedia
- F = floating point or SIMD
- B = branch

Some instructions can be used in multiple slot types, and the disassembler will stick a suffix (known as a *completer*) to disambiguate them. For example, there are five different `nop` instructions, one for each slot type: `nop.m`, `nop.i`, `nop.a`, `nop.f`, and `nop.b`. When reading code, you don't need to worry too much about slotting. You can assume that the compiler did it correctly; otherwise it wouldn't have disassembled properly! (For the remainder of this series, I will tend to omit completers if their sole purpose is to disambiguate a slot type.)

If you are debugging unoptimized code, you may very well see a lot of `nop`s because the compiler didn't bother trying to optimize slot usage.

Another thing that bundles encode is the placement of what are known as *stops*. A stop is used to indicate that the instructions after the stop depend on instructions before the stop. For example, if you had the following sequence of instructions

```
mov r3 = r2
add r1 = r2, r4 ;;
add r2 = r1, r3
```

there is no dependency between the first two instructions; they can execute in parallel. However, the third instruction cannot execute until the first two have completed. The compiler therefore inserts a stop after the second instruction, which is represented by a double-semicolon.

A sequence of instructions without any stops is known as an *instruction group*. (There are other things that can end an instruction group, but they aren't important here.) As noted above, the instructions in an instruction group may not have any dependencies among them. This allows the processor to execute them in parallel. (This is an example of how the processor relies on the compiler: By making it the compiler's responsibility to ensure that there are no dependencies within an instruction group, the processor can avoid having to do its own dependency analysis.)

There are some exceptions to the rule against having dependencies within an instruction group:

- A branch instruction is allowed to depend on a predicate register and/or branch register set up earlier in the group.
- You are allowed to use the result of a successful `ld.c` without an intervening stop. We'll learn more about `ld.c` when we discuss explicit speculation.
- Comparison instructions `.and`, `.andcm`, `.or`, and `.orcm` are allowed to combine with others of the same type into the same targets. (In other words, you can combine two `.and`s, but not an `.and` and an `.or`.)
- You are allowed to write to a register after a previous instruction reads it. (With rare exceptions.)
- Two instructions in the same group cannot write to the same register. (With the exception of combined comparisons noted above.)

There are a lot of fine details in the rules, but I'm ignoring them because they are of interest primarily to compiler-writers. The above rules are to give you a general idea of the sorts of dependencies that can exist within an instruction group. (Answer: Not much.)

It does highlight that writing ia64 assembly by hand is exceedingly difficult because you have to make sure every triplet of instructions you write matches a valid template in terms of slots and stops, and you have to ensure that the instruction groups do not break the rules.

Next time, we'll look at the calling convention.

[1] There are two templates which are special in that they encode only two slots rather than three. The first slot is the normal 41 bits, but the second slot is a double-wide 82 bits. The double-wide slot is used by a few special-purpose instructions we will not get into.

Raymond Chen

Follow