The Itanium processor, part 5: The GP register, calling functions, and function pointers

devblogs.microsoft.com/oldnewthing/20150731-00

July 31, 2015

Raymond Chen

We saw a brief mention of the *gp* register last time, where we saw it used when we calculated the address of a global variable.

The only addressing mode supported by the Itanium processor is register indirect (possibly with post-increment). There is no absolute addressing mode. If you want to access a global variable, you need to calculate its address, and the convention for this is that the *gp* register points to the module's global variables. If you want to access a global variable stored at offset *n* in the global data segment, you do it in two steps:

```
addl    r30 = n, gp ;;    // r30 -> global variable
ld4     r30 = [r30]       // load 4 bytes from the global variable
```

The name *gp* stands for *global pointer* since it is the pointer used to access global variables. (Note that since immediates are signed, the range of values of *n* is −2MB to +2MB.)

Those of you familiar with the PowerPC will recognize this model, since it is very similar to the Table of Contents model, except that Itanium uses a single table of contents for the entire module, as opposed to the PowerPC which gives each function its own table of contents.

The Itanium *addl* instruction is limited to a 22-bit immediate, which provides a reach of 4MB. This means that the above pattern is viable only for 4MB of global variables. Since some modules have more than 4MB of global data, the compiler separates global data into two categories, *large* and *small*. Small data objects are stored directly in the global data segment, but large data objects are not. Instead, the large data object is placed outside the global data segment, and all that is placed in the global data segment is a pointer to the large object. This means that accessing a large object actually takes three instructions.

```
addl    r30 = n, gp ;; // r30 -> global variable forwarder
ld8     r30 = [r30] ;; // r30 -> global variable
ld4     r30 = [r30]    // load 4 bytes from the global variable
```

We see that it is vitally important that the *gp* register be set properly. Otherwise, the code has no idea where its global variables are. The Itanium calling convention says that on entry to a function, the *gp* register must be set to that function's global pointer.

Okay, so if you're going to call a function, how do you know what global pointer it expects?

Since all functions in the same module share the same global variables, the answer is easy if you are calling a function within the same module: You don't need to do anything special with *gp*, since the caller's *gp* is the same as the callee's *gp*. You also don't need to perform an indirect call; you know where the target is and can use a direct `br.call OtherFunction`.[1]

On the other hand, if you are calling a function through a function pointer, then the target of the call might belong to another module. How are you supposed to know what the target function wants *gp* to be?

The answer is that on Itanium, a function pointer is not the address of the first instruction. Rather, it is a pointer to a structure containing two pointers. The first pointer in the structure points to the first instruction of the target function. The second pointer is the target function's *gp*. Therefore, calling a function through a function pointer looks like this:

```
// suppose the function pointer is in r30
ld8     r31 = [r30], 8 ;;       // get the function address
                                // then add 8 to r30
ld8     gp = [r30]              // get the function's gp
mov     b6 = r31                // move to branch register
br.call.dptk.many rp = b6 ;;    // call function in b6
or      gp = r41, r0            // gp = r41 OR 0 = r41
```

First, we load the address of the first instruction into the *r31* register, using a post-increment addressing mode so that *r30* after the instruction points to the callee's *gp*.

Next, we load the *gp* register with the caller's *gp*. Simultaneously, we move *r31* to *b6* so that we can use it as the target of the *br.call*. (Branch registers cannot be the target of a *ld8* instruction, which is why we needed to use *r31* as a middle-man.)

Now that *gp* is set up properly, we can call the function through the branch register.

After the call returns, the *gp* register is now whatever value is left over by the function we called. We need to set *gp* to the current function's global pointer, which for the sake of example we'll assume had been saved in the *r41* register.

There's yet another wrinkle: The naïve imported function. In the case of an imported function not declared with the `dllimport` attribute, the compiler doesn't know that the function is imported. It acts as if the function is part of the current module. On x86, this is simulated by making a stub function which jumps to the real (imported) function. On Itanium, the same thing is done, with a stub function that looks like this:

```
.ImportedFunction:
        addl    r30 = n, gp ;;      // r30 -> function descriptor
        ld8     r31 = [r30], 8;;    // get the function address
                                    // then add 8 to r30
        ld8     gp = [r30]          // get the function's gp
        mov     b6 = r31            // move to branch register
        br.cond.sptk.many b6 ;;     // jump there
```

The stub function loads the *gp* register with the value expected by the imported function then jumps to the imported function. Unconditional computed jumps are encoded as conditional jumps where the qualifying predicate is *p0*, which is always true.

The possibility that any function is really a stub function for an imported function this creates a problem for the compiler: Since any function could be an imported function in disguise, the compiler must assume that *any function* is potentially imported and therefore may result in the *gp* register being trashed. Therefore, the compiler needs to restore the *gp* register after *any* function call.

Now, the above pessimistic assumption can be relaxed if the compiler has other information available to it. For example, if the function being called is in the same translation unit, then the compiler can see by inspection that the target function is not a stub and therefore can elide the restoration of *gp*. Similarly, if link-time code generation is enabled, then the linker can see all the code in the module and see whether the target function is a stub or a real function.

Exercise: How does tail-call elimination affect this optimization?

Bonus reading: *Programming for 64-bit Windows* which spends nearly all its time talking about the *gp* register.

[1] The direct call instruction has a reach of 16MB, so if the function you want to call is too far away, the linker redirects the *br.call* to a stub function which in turn jumps to the final destination.

```
    br.call.dptk.many stub_for_OtherFunction
...

stub_for_OtherFunction:
    ... jump to OtherFunction ...
```

You have a few options for jumping to the function.

If the stub is within 16MB of the target, it can use a *br.cond* direct jump:

```
stub_for_OtherFunction:
    br.cond.sptk.many OtherFunction
```

The stub can load the target address from the data segment and use an indirect jump:

```
stub_for_OtherFunction:
    addl r3 = n, gp ;;  // look up the function address
    ld8  r3 = [r3] ;;   // fetch it
    mov  b6 = r3 ;;     // prepare to jump there
    br.cond.sptk.many b6 ;; // and off we go
```

> The stub can load the target address offset from data stored in the code segment, then apply the offset to the current instruction pointer to determine the target:

```
stub_for_OtherFunction:
    mov  r3 = iip ;;    // get current location
    addl r3 = n, r3 ;;  // find the offset
    ld8  r2 = [r3] ;;   // load the offset
    addl r2 = r2, r3 ;; // apply to current location
    mov  b6 = r2 ;;     // prepare to jump there
    br.cond.sptk.many b6 ;; // and off we go
```

This last case is tricky because the Itanium conventions forbid relocations in the code segment; all code is position-independent. Therefore, the data in the code segment must not be relocatable. We work around this by storing an offset rather than the absolute address and applying the offset at runtime.

Raymond Chen

Follow