The PowerPC 600 series, part 8: Control transfer

devblogs.microsoft.com/oldnewthing/20180815-00

August 15, 2018

Raymond Chen

The PowerPC 600 series has a few types of control transfer instructions. Let's look at direct branches first.

```
    b       target          ; branch to target
    bl      target          ; branch to target and link
```

The direct branch instructions perform an unconditional relative branch to the target. It has a reach of ±32MB. All the "… and link" instructions set the *lr* register to the return address (the instruction after the branch). This happens even for conditional branches when the branch is not taken.

There are also absolute versions of these instructions:

```
    ba      target          ; branch to target (absolute form)
    bla     target          ; branch to target and link (absolute form)
```

The absolute versions treat the displacement as an absolute address rather than as a displacement from the current instruction pointer. These are not useful in Windows NT, but could be useful in embedded systems.

Things get exciting when you look at the conditional branches. Formally, they are written as

```
    bc      BO, BI, target  ; branch conditional
    bcl     BO, BI, target  ; branch conditional and link
```

Conditional branch instructions have a reach of only ±32KB. There are also absolute variants `bca` and `bcla` which treat the displacement as an absolute address, allowing conditional branches to the top and bottom 32KB of address space. Again, absolute addressing is not that useful in Windows NT.

The magical *BO* and *BI* parameters describe the condition to be tested. You can optionally decrement the *ctr* register and check if the result is zero or nonzero.[1] You can also optionally check if a specific bit in the *cr* register is set (true) or clear (false), and sometimes you can provide a static prediction hint. The following combinations are valid:

Decrement *ctr*?	**Test a bit in *cr*?**	**Prediction hint**	***BO***	**Mnemonic**
Yes, test for nonzero	No		16	`dnz`
Yes, test for nonzero	No	Not taken	24	`dnz-`
Yes, test for nonzero	No	Taken	25	`dnz+`
Yes, test for nonzero	Test for false		0	`dnzf`
Yes, test for nonzero	Test for true		8	`dnzt`
Yes, test for zero	No		18	`dz`
Yes, test for zero	No	Not taken	26	`dz-`
Yes, test for zero	No	Taken	27	`dz+`
Yes, test for zero	Test for true		10	`dzt`
Yes, test for zero	Test for false		2	`dzf`
No	Test for false		4	`f`
No	Test for false	Not taken	6	`f-`
No	Test for false	Taken	7	`f+`
No	Test for true		12	`t`
No	Test for true	Not taken	14	`t-`
No	Test for true	Taken	15	`t+`
Unconditional		Taken	20	

Any *BO* values not in the above table are reserved for future use and should be avoided if you know what's good for you.

A static prediction hint overrides any internal branch prediction algorithm, so you'd better have very high confidence that your hint is correct.

These mnemonics save you from having to memorize the *BO* numbers.

```
    bxx     BI, target  ; branch conditional
    bxxl    BI, target  ; branch conditional and link
```

Except that if the mnemonic ends in a `+` or `-`, then the prediction hint goes at the very end. For example, "branch if false and link, predict not taken" is `bfl-`.

The bit index *BI* can be written as a number, but as we saw when we learned about condition registers, you can combine the condition register bit mnemonics with with the *cr#* mnemonics to produce a reference to a condition bit. For example, `4*cr2+gt` means "The *gt* bit in the *cr2* condition register." And since the numeric value of *cr0* is zero, you can omit `4*cr0+`, which results in some surprisingly readable results like

```
    bt      eq, target  ; branch if eq is set in cr0
```

The assembler goes one step further and provides a few combination mnemonics:²

Branch and condition	Mnemonic	Meaning
`bt lt`	`blt`	Branch if less than
`bt gt`	`bgt`	Branch if greater than
`bt eq`	`beq`	Branch if equal
`bt so`	`bso`	Branch if summary overflow
`bf lt`	`bnl`	Branch if not less than
`bf gt`	`bng`	Branch if not greater than
`bf eq`	`bne`	Branch if not equal
`bf so`	`bns`	Branch if not summary overflow

The mnemonics can separate the condition bit from the condition register, so you can get

```
    beq     cr4, target  ; branch if eq is set in cr4
```

Okay, the next type of branch instruction is the computed jump.

```
    bcctr   BO, BI, BH   ; branch conditional to address in ctr
    bcctrl  BO, BI, BH   ; branch conditional to address in ctr and link

    bclr    BO, BI, BH   ; branch conditional to address in lr
    bclrl   BO, BI, BH   ; branch conditional to address in lr and link
```

You are not allowed to use any of the "decrement *ctr*" branch operations with the `bcctr` or `bcctrl` instructions because shame on you for even thinking about trying it.

The *BO* and *BI* codes follow the same rules as above, and the assembler provides mnemonics for various combinations. If you go to PowerPC reference materials, you'll see horrid tables that look like some sort of dystopian declension table from a long-forgotten Slavic language.

In this hypothetical language, `bdnztlrl` means something like "branch on odd-numbered Thursdays," I guess. (Okay, it actually means "branch, after decrementing `ctr`, if the result is nonzero, and if the condition bit is true, to the address in the `lr` register, and link.")

The *BH* field provides a hint for branch prediction, primarily whether the branch target is likely to be the same as the previous time the branch was encountered. Branches through an import table are likely to be the same each time. Branches through a vtable could also use this hint if the method is being dispatched from the same object in a loop. (The vtable is unlikely to change during the loop.)

The processor optimizes on the assumption that `bctr` is a computed jump and `blr` is a subroutine return,3 although the *BH* hints can tweak those assumptions. Furthermore, Windows NT *requires* that non-leaf subroutine returns be encoded exclusively as `blr`. You are not allowed to pull fancy tricks like `beqlr` to perform a conditional subroutine return. This is not a significant problem in practice because there's usually other stuff that needs to be done as part of the function epilogue. Adding this rule makes the exception unwinding code easier.

For the same reason, the conditional versions of the "and link" branches are mostly useless in practice because even if you can conditionalize the link, you still prepared the function call unconditionally. You might have been better off just branching over the function call entirely.

Okay, so great, you have these instructions that operate on the *lr* and *ctr* registers, but how do you actually get values in and out of them?

```
    mflr    rt              ; rt = lr
    mfctr   rt              ; rt = ctr

    mtlr    rs              ; lr = rs
    mtctr   rs              ; ctr = rs
```

The "move from/to *lr*/*ctr*" instructions let you move values into and out of the *lr* and *ctr* registers. (Like `mfxer` and `mtxer`, these are actually shorthand for `mfspr` and `mtspr` with the appropriate magic number for *lr* or *ctr*.)

In practice, the first instruction of a non-leaf function is `mflr r0` to save the return address, and when it's ready to return, it will do a `mtlr r0` to load up the return address in preparation for the `blr`. This is pretty much the only thing the Microsoft compiler uses the *r0* register for: Transferring the return address in and out of *lr*.

But wait, I'm getting ahead of myself. I promised to talk about the table of contents, so let's do that next time.

Bonus chatter: PowerPC mnemonics are so absurd that there was even a short-lived parody twitter account for them. Now that you've learned most of the instructions, you may understand some of the more insidey jokes, like

> mscdfr – Means Something Completely Different For r0
>
> — PowerPC Instructions (@ppcinstructions) January 21, 2015

[1] Note that even if you loaded a 64-bit value into the *ctr* register (because you detected that you had a 64-bit-capable processor), the test for zero or non-zero is performed only against the least-significant 32 bits of the *ctr* register when the processor is in 32-bit mode (which is what Windows NT uses).

[2] The assembler also provides `bge` (branch if greater than or equal to) as an alias for `bnl` (branch if not less than). I think that's misleading, because `bge` suggests that the test checks two bits (*gt* and *eq*) and branches if either is set. But in fact it checks whether *lt* is clear. Now, if the condition register was set by a comparison, then the two cases are equivalent, but if you have been playing games with condition register flags, you can get into states where the trichotomy of numbers breaks down.

[3] The return address predictor gives the processor the ability to start speculating instructions at the return address even before you move the return address into the *lr* register!

Raymond Chen

Follow