The PowerPC 600 series, part 2: Condition registers and the integer exception register

devblogs.microsoft.com/oldnewthing/20180807-00

August 7, 2018

Raymond Chen

The integer exception register *xer* contains a bunch of stuff, but the ones that are relevant to us are

Bit	Name
0	Summary overflow
1	Overflow
2	Carry

Some instructions update the overflow and summary overflow bits in the *xer* register. When those instructions are executed, the overflow bit is updated to represent whether the operation resulted in a signed overflow. The summary overflow bit accumulates all the overflow bits since it was last explicitly reset. This lets you perform a series of arithmetic operations and then test a single bit at the end to see if an overflow occurred anywhere along the way.

Some instructions consume and/or target the carry bit in *xer*. We'll discuss how carry works when we get to integer arithmetic.

Each of the *cr#* condition registers consists of four bits, numbered from most signficant to least significant.

Bit	Name	Mnemonic
0	Less than	*lt*
1	Greater than	*gt*
2	Equal to	*eq*

3	Summary overflow	*so*

For convenience, the assembler predefines the constants *lt*, *gt*, *eq*, and *so* to represent their respective bit numbers.

The `cmp` family of instructions compare two values and write the result to a condition register.

```
cmpw    crd, ra, rb     ; crd = compare ( int32_t)ra with ( int32_t)rb
cmpwi   crd, ra, imm16  ; crd = compare ( int32_t)ra with ( int16_t)imm16
cmplw   crd, ra, rb     ; crd = compare (uint32_t)ra with (uint32_t)rb
cmplwi  crd, ra, imm16  ; crd = compare (uint32_t)ra with (uint16_t)imm16
```

You can compare two registers, or you can compare a register with an immediate, and you can choose whether the comparison is signed or unsigned. (Recall that the `l` stands for *logical*.) For example:

```
cmpw    cr3, r0, r1     ; cr3 = compare r0 with r1 as signed values
```

The *lt*, *gt*, and *eq* bits are set according to the result of the comparison, and the *so* bit receives a copy of the current summary overflow bit in *xer*.

If you do not specify a destination comparison register, it defaults to *cr0*:

```
cmpw    ra, rb          ; cr0 = compare ( int32_t)ra with ( int32_t)rb
cmpwi   ra, imm16       ; cr0 = compare ( int32_t)ra with ( int16_t)imm16
cmplw   ra, rb          ; cr0 = compare (uint32_t)ra with (uint32_t)rb
cmplwi  ra, imm16       ; cr0 = compare (uint32_t)ra with (uint16_t)imm16
```

As we'll see later, some arithmetic instructions implicitly update *cr0* by comparing the computed result against zero. (Similarly, some floating point operations implicitly update *cr1*.) When performed as part of an arithmetic instruction, the comparison is always performed as a signed comparison, even if the instruction's underlying operation was unsigned.

If you combine an update of *cr0* with an arithmetic operation, the *so* bit is a copy of the summary overflow bit in the *xer* register at the end of the instruction. That means that if an arithmetic operation requests both *cr0* and *xer* to be updated, the *xer* register is updated first, and then the summary overflow bit from *xer* is copied to the *so* bit in *cr0*. That means that the *so* bit in *cr0* captures whether a signed overflow occurred in any overflow-detecting operation up to and including the current one.

The Microsoft compiler tends to prefer to target *cr6* and *cr7* in its comparison instructions. It doesn't make much difference to the processor, but I suspect the compiler tries to avoid *cr0* so that it doesn't conflict with the use of *cr0* by the arithmetic instructions.

```
    mcrxr  crd              ; crd = first four bits of xer
```

The “move to condition register from *xer*” instruction copies the summary overflow, overflow, and carry bits from the *xer* register to the specified condition register, and then it clears the bits from *xer*.

No, I don’t know why they left the “e” out of the opcode.

This is how you reset the summary overflow.[1]

```
    mtxer  ra               ; xer = ra
    mfxer  rd               ; rd = xer
```

These instructions[2] move to/from the *xer* register. They are another way to clear the *xer* register, or to set it to a particular initial state.

There are a good number of bitwise operations that combine two condition register bits and store the result into a third condition register bit. These let you build boolean expressions out of condition registers.

```
    crand   bd, ba, bb  ; cr[bd] =   cr[ba] &  cr[bb]
    cror    bd, ba, bb  ; cr[bd] =   cr[ba] |  cr[bb]
    crxor   bd, ba, bb  ; cr[bd] =   cr[ba] ^  cr[bb]
    crnand  bd, ba, bb  ; cr[bd] = !(cr[ba] &  cr[bb])
    crnor   bd, ba, bb  ; cr[bd] = !(cr[ba] |  cr[bb])
    creqv   bd, ba, bb  ; cr[bd] = !(cr[ba] ^  cr[bb])
    crandc  bd, ba, bb  ; cr[bd] =   cr[ba] & !cr[bb] "and complement"
    crorc   bd, ba, bb  ; cr[bd] =   cr[ba] | !cr[bb] "or complement"
```

Remember that the PowerPC numbers bits from most significant to least significant, so bit zero is the high-order bit.

To save you from having to memorize all the bit numbers, the assembler lets you write *cr0* to mean 0, *cr1* to mean 1, and so through *cr7* which means 7. Combined with the constants for the four bits in the condition register, this lets you write

```
    crand   4*cr3+eq, 4*cr2+lt, 4*cr6+gt ; cr3[eq] = cr2[lt] & cr6[gt]
```

instead of the instruction only a processor’s mother could love:

```
    crand   14, 8, 25                    ; cr3[eq] = cr2[lt] & cr6[gt]
```

There are also special instruction for transferring between *cr* and a general-purpose register.

```
    mfcr    rt          ; rt = cr
    mtcrf   mask, ra    ; cr = ra (selected by mask)
```

The mask is an 8-bit immediate. If a bit is set, then the corresponding *cr#* is copied from the corresponding bits of *ra*. For example, 128 means "Copy the top four bits of *ra* into *cr0*, and leave all the other condition registers alone." Recall that the PowerPC counts bits from most significant to least significant, so *cr0* is stored in the highest-order four bits.

The assembler provides synthetic instructions for various special cases of the above operations:

```
    creqv   bd, bd, bd  ; crset   bd           ; cr[bd]  = 1
    crxor   bd, bd, bd  ; crclr   bd           ; cr[bd]  = 0
    cror    bd, ba, ba  ; crmove  bd, ba       ; cr[bd]  = cr[ba]
    crnor   bd, ba, ba  ; crnot   bd, ba       ; cr[bd]  = !cr[ba]
    mtcr    ra          ; mtcrf   255, ra      ; cr = ra
```

Here's an example of how these boolean operations could be used:

```
    cmpw    cr2, r4, r5 ; compare r4 with r5, put result in cr2
    cmpw    cr3, r6, r7 ; compare r6 with r7, put result in cr3
    crandc  4*cr0+eq, 4*cr2+gt, 4*cr4+eq ; cr0[eq] = cr2[gt] & !cr4[eq]
    beq     destination ; jump if r4 > r5 && r6 != r7
```

We perform two comparison operations and put the results into *cr2* and *cr3*. We then perform a boolean "and not" operation that calculates

```
    cr0[eq] = (r4 > r5) & !(r6 == r7)
            = (r4 > r5) & (r6 != r7)
```

The result is placed into the *eq* position of *cr0*, which makes it a perfect place to be the branch condition of the `beq` instruction.

The traditional way of doing this on processors that don't have these fancy condition register operations is to perform a test and a conditional branch, then another test and another conditional branch. Combining the results of the test and performing a single branch means that the entire sequence consumes only one slot in the branch predictor. This leaves more slots free to predict other branches, and the single slot this sequence does consume can predict the final result, which might be easier to predict than the individual pieces. (For example, the test might be validating that a parameter is one of two valid values. The parameter is almost always valid, even though one might not be able to predict which of the two valid values it is at any particular time.)

Fabian Giesen notes that in practice, you don't get to perform this optimization as often as you'd like because of short-circuiting rules in many programming languages. Under those rules, this optimization works only if the second term can be evaluated without any risk of taking any exceptions (or if the language permits you to take an exception anyway, say, because any exception would be the result of undefined behavior).

I have yet to see the Microsoft C compiler for PowerPC perform this optimization. It just does things the conventional way. But that may just be because I haven't encountered a situation where the optimization is even possible. (Also, because I'm studying code from Windows NT 3.51, and compiler technology was not as advanced back then.)

Okay, next time we'll start doing some arithmetic.

[1] You might have noticed that there are only three interesting bits in *xer* but room for four bits in a condition register. The last bit is undefined. Usually, you don't care much about the bits that got transferred; the main purpose of the instruction is its side effect of clearing the summary overflow.

[2] These instructions are actually special cases of the `mtspr` and `mfspr` instructions which move to/from a special register. The *xer* register is formally register *spr1*, so the `mtxer` and `mfxer` instructions are technically synthetic instructions.

```
    mtspr   1, ra               ; spr1 = ra
    mfspr   1, rd               ; rd = spr1
```

Raymond Chen

Follow