The PowerPC 600 series, part 3: Arithmetic

devblogs.microsoft.com/oldnewthing/20180808-00

August 8, 2018

Raymond Chen

Before we start with arithmetic, we need to have a talk about carry.

The PowerPC uses true carry for both addition and subtraction. This is different from the x86 family of processors, for which the carry flag is actually a borrow bit when used in subtraction. You can read more about the difference on Wikipedia. There are some instructions which perform a combined addition and subtraction, and in that case, the only sane choice is to use true carry. (If you had chosen carry as borrow, then it wouldn't be clear whether the final carry bit represented the carry from the addition or the borrow from subtraction.)

To emphasize the fact that the PowerPC uses true carry, I will rewrite all subtractions as additions, taking advantage of the twos complement identity

```
-x = ~x + 1
```

Okay, now we can do some arithmetic. Let's start with addition.

```
add     rd, ra, rb      ; rd = ra + rb
add.    rd, ra, rb      ; rd = ra + rb, update cr0
addo    rd, ra, rb      ; rd = ra + rb, update         XER overflow bits
addo.   rd, ra, rb      ; rd = ra + rb, update cr0 and XER overflow bits
```

These instructions add two source registers and optionally update the *xer* register to capture any possible overflow (by appending an `o`), and also optionally update the *cr0* register to reflect the sign of the result and any summary overflow (by appending a period).

I don't know what they were thinking, using an easily-overlooked mark of punctuation to carry important information.

There is also a version of the above instruction that takes a signed 16-bit immediate:

```
addi    rd, ra/0, imm16 ; rd = ra/0 + (int16_t)imm16
```

Note that this variant does not accept `o` or `.` suffixes.

The *ra/0* notation means “This can be any general purpose register, but if you ask for *r0*, you actually get the constant zero.” The register *r0* is weird like that. Sometimes it stands for itself, but sometimes it reads as zero. As a result, the *r0* register isn’t used much.

The assembler lets you write *r0* through *r31* as synonyms for the integers 0 through 31, so the following are equivalent:

```
    add     r3, r0, r4      ; r3 = r0 + r4
    add      3,  0,  4      ; r3 = r0 + r4
    add     r3, r0,  4      ; r3 = r0 + r4
```

This can get very confusing. That last example sure looks like you’re setting *r3* to *r0* plus 4, but it’s not. The 4 is in a position where a register is expected, so it actually means *r4*.

Similarly, you might think you’re adding an immediate to *r0* when you write

```
    addi    r3, r0, 256     ; r3 = r0 + 256, right?
```

but nope, the value of 0 as the second operand to `addi` is interpreted as the constant zero, not register number zero.

Fortunately, the Windows disassembler always calls registers by their mnemonic rather than by number.

Wait, we’re not done with addition yet.

```
    ; add and set carry
    addc    rd, ra, rb      ; rd = ra + rb, update carry
    addc.   rd, ra, rb      ; rd = ra + rb, update carry and cr0
    addco   rd, ra, rb      ; rd = ra + rb, update carry         and XER overflow
bits
    addco.  rd, ra, rb      ; rd = ra + rb, update carry and cr0 and XER overflow
bits
```

The “add and set carry” instructions act like the corresponding regular add instructions, except that the also update the carry bit in *xer* based on whether a carry propagated out of the highest-order bit.

```
    ; add extended
    adde    rd, ra, rb      ; rd = ra + rb + carry, update carry
    adde.   rd, ra, rb      ; rd = ra + rb + carry, update carry and cr0
    addeo   rd, ra, rb      ; rd = ra + rb + carry, update carry         and XER
overflow bits
    addeo.  rd, ra, rb      ; rd = ra + rb + carry, update carry and cr0 and XER
overflow bits
```

The “add extended” instructions act like the corresponding “add and set carry” instructions, except that they also add 1 if the carry bit was set. This makes multiword addition convenient.

```
    ; add minus one extended
    addme   rd, ra          ; rd = ra + carry + ~0, update carry
    addme.  rd, ra          ; rd = ra + carry + ~0, update carry and cr0
    addmeo  rd, ra          ; rd = ra + carry + ~0, update carry         and XER
overflow bits
    addmeo. rd, ra          ; rd = ra + carry + ~0, update carry and cr0 and XER
overflow bits
```

The “add minus one extended” instruction is like “add extended” except that the second parameter is hard-coded to −1. I wrote `~0` instead of −1 to emphasize that we are using true carry. (This is the combined addition-and-subtraction instruction I alluded to at the top of the article. It adds carry and then subtracts one.) **Added**: As commenter Neil noted below, through the magic of true carry, this is the same as “subtract zero extended”, which makes it handy for multiword arithmetic.

```
    ; add zero extended
    addze   rd, ra          ; rd = ra + carry, update carry
    addze.  rd, ra          ; rd = ra + carry, update carry and cr0
    addzeo  rd, ra          ; rd = ra + carry, update carry         and XER overflow
bits
    addzeo. rd, ra          ; rd = ra + carry, update carry and cr0 and XER overflow
bits
```

The “add zero extended” instruction is like “add extended” except that the second parameter is hard-coded to zero.

And then there are some instructions that take signed 16-bit immediates:

```
    ; add immediate shifted
    addis   rd, ra/0, imm16  ; rd = ra/0 + (imm16 << 16)

    ; add immediate and set carry
    addic   rd, ra, imm16    ; rd = ra + (int16_t)imm16, update carry

    ; add immediate and set carry and update cr0
    addic.  rd, ra, imm16    ; rd = ra + (int16_t)imm16, update carry and cr0
```

Phew, that was addition. There are also subtraction instructions, which should look mostly familiar now that you’ve seen addition.

```
    ; subtract from
    subf    rd, ra, rb      ; rd = ~ra + rb + 1
    subf.   rd, ra, rb      ; rd = ~ra + rb + 1, update cr0
    subfo   rd, ra, rb      ; rd = ~ra + rb + 1, update         XER overflow bits
    subfo.  rd, ra, rb      ; rd = ~ra + rb + 1, update cr0 and XER overflow bits

    ; subtract from and set carry
    subfc   rd, ra, rb      ; rd = ~ra + rb + 1, update carry
    subfc.  rd, ra, rb      ; rd = ~ra + rb + 1, update carry and cr0
    subfco  rd, ra, rb      ; rd = ~ra + rb + 1, update carry         and XER
overflow bits
    subfco. rd, ra, rb      ; rd = ~ra + rb + 1, update carry and cr0 and XER
overflow bits

    ; subtract from extended
    subfe    rd, ra, rb     ; rd = ~ra + rb + carry, update carry
    subfe.   rd, ra, rb     ; rd = ~ra + rb + carry, update carry and cr0
    subfeo   rd, ra, rb     ; rd = ~ra + rb + carry, update carry         and XER
overflow bits
    subfeo.  rd, ra, rb     ; rd = ~ra + rb + carry, update carry and cr0 and XER
overflow bits

    ; subtract from minus one extended
    subfme   rd, ra         ; rd = ~ra + carry + ~0, update carry
    subfme.  rd, ra         ; rd = ~ra + carry + ~0, update carry and cr0
    subfmeo  rd, ra         ; rd = ~ra + carry + ~0, update carry         and XER
overflow bits
    subfmeo. rd, ra         ; rd = ~ra + carry + ~0, update carry and cr0 and XER
overflow bits

    ; subtract from zero extended
    subfze   rd, ra         ; rd = ~ra + carry, update carry
    subfze.  rd, ra         ; rd = ~ra + carry, update carry and cr0
    subfzeo  rd, ra         ; rd = ~ra + carry, update carry         and XER overflow
bits
    subfzeo. rd, ra         ; rd = ~ra + carry, update carry and cr0 and XER overflow
bits

    ; subtract from immediate and set carry
    subfic  rd, ra, imm16   ; rd = ~ra + (int16_t)imm16 + 1, update carry
```

Note that the instruction is “subtract from”, not “subtract”. The second operand is subtracted from the third operand; in other words, the two operands are backwards. Fortunately, the assembler provides a family of synthetic instructions that simply swap the last two operands:

```
    subf    rd, rb, ra      ; sub  rd, ra, rb
    ; similarly "sub.", "subo", and "subo.".

    subfc   rd, rb, ra      ; subc rd, ra, rb
    ; similarly "subc.", "subco", and "subco.".
```

Second problem is that there is no `subfis` to subtract a shifted immediate, nor is there `subfic.` to update flags after subtracting from an immediate. But the assembler can synthesize those too:

```
addi    rd, ra/0, -imm16 ; subi   rd, ra/0, imm16
addis   rd, ra/0, -imm16 ; subis  rd, ra/0, imm16
addic   rd, ra, -imm16   ; subic  rd, ra, imm16
addic.  rd, ra, -imm16   ; subic. rd, ra, imm16
```

PowerPC's use of true carry allows this trick to work while still preserving the semantics of carry and overflow.

We wrap up with multiplication and division.

```
; multiply low immediate
mulli   rd, ra, imm16    ; rd = (int32_t)ra * (int16_t)imm16

; multiply low word
mullw   rd, ra, rb       ; rd = (int32_t)ra * (int32_t)rb
; also "mullw.", "mullwo", and "mullwo.".

; multiply high word
mulhw   rd, ra, rb       ; rd = ((int32_t)ra * (int32_t)rb) >> 32
; also "mulhw."

; multiply high word unsigned
mulhwu  rd, ra, rb       ; rd = ((uint32_t)ra * (uint32_t)rb) >> 32
; also "mulhwu."
```

The "multiply low" instructions perform the multiplication and return the low-order 32 bits. The "multiply high" instructions return the high-order 32 bits.

Finally, we have division:

```
; divide word
divw    rd, ra, rb       ; rd = (int32_t)ra ÷ (int32_t)rb
; also "divw.", "divwo", and "divwo.".

; divide word unsigned
divwu   rd, ra, rb       ; rd = (uint32_t)ra ÷ (uint32_t)rb
; also "divwu.", "divwuo", and "divwuo.".
```

If you try to divide by zero or (for `divw`) if you try to divide `0x80000000` by −1, then the results are garbage, and if you used the `o` version of the instruction, then the overflow flag is set. No trap is generated. (If you didn't use the `o` version, then you get no indication that anything went wrong. You just get garbage.)

There is no modulus instruction. If you want to get the remainder, take the quotient, multiple it by the divisor, and subtract it from the dividend.

Okay, that was arithmetic. Next up are the bitwise logical operators and combining arithmetic and logical operators to load constants.

Bonus snark: For a reduced instruction set computer, it sure has an awful lot of instructions. And we haven't even gotten to control flow yet.

Raymond Chen

Follow