The PowerPC 600 series, part 5: Rotates and shifts

devblogs.microsoft.com/oldnewthing/20180810-00

August 10, 2018

Raymond Chen

The Swiss army knife instruction of the PowerPC 600 series instruction set is `rlwinm`, which stands for "rotate left word immediate and mask."

```
rlwinm  rd, ra, imm5a, imm5b, imm5c
rlwinm. rd, ra, imm5a, imm5b, imm5c ; also updates cr0
```

This instruction does everything except wash your floor.

First it takes the current value of the *ra* register and rotates it left by *imm5a* bits. Then it keeps all the bits *imm5b* to *imm5c*, inclusive, and clears all the other bits. If *imm5b* is greater than *imm5c*, the bits to keep wrap around! (Don't forget that bit zero is the most significant bit.) The result of all this is placed in the *rd* register.

Let's take a few examples.

```
rlwinm  rd, ra, 5, 6, 20
```

Here, we take *ra*, rotate it left by 5 bits, and then keep bits 6 through 20. Since bit 0 is the most significant bit, then that means that the mask is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
`0`				`3`				`F`				`F`				`F`				`8`	

This comes out to `0x03FFF800`. Therefore, the result of the operation is

```
rd = rotl(ra, 5) & 0x03FFF800
```

The other case is where the bit count wraps around:

```
rlwinm  rd, ra, 5, 20, 6
```

This time, we start by setting bit 20, and continue to the end of the word, and then wrap around and start setting bits starting at bit 0, and then finally stop when we get to bit 6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
F				E				0				0				0				F	

The net result is therefore

```
rd = rotl(ra, 5) & 0xFE000FFF
```

The Windows debugger for PowerPC was not in use for very long, and consequently its disassembler isn't particularly advanced. But one thing it does do for you is unpack the last two parameters and tell you what the final mask is.

```
10ae222c 54642d0c rlwinm  r4, r3,5,20,6           MASK=0xfe000fff
```

(Yes, the disassembler can't make up its mind whether it puts a space after a comma or not. Like I said, the debugger didn't last long enough to go through multiple iterations of polish.)

With the `rlwinmi` instruction, we can build many things. (In all examples, bit index arithmetic wraps around at 32.)

- Rotate left by *n* bits:

  ```
  rlwinm  rd, ra, n, 0, 31
  ```

  Rotate left by *n* bits, and specify a mask that starts at bit 0 and ends at bit 31. This mask is `0xFFFFFFFF`, which means that no bits get cleared.

- Rotate right by *n* bits:

  ```
  rlwinm  rd, ra, 32 - n, 0, 31
  ```

  Rotate left by 32 – *n* bits, and specify a mask that starts at bit 0 and ends at bit 31. This mask is `0xFFFFFFFF`, which means that no bits get cleared.

- Shift left by *n* bits:

  ```
  rlwinm  rd, ra, n, 0, 31 - n
  ```

  Rotate left by *n* and clear the rightmost *n* bits. In Windows NT, the most common version of this is

  ```
  rlwinm  rd, ra, 2, 0, 29
  ```

  which shifts a value left two places. This multiplies it by four, which is a common operation when indexing an array of pointers.

- Shift right by *n* bits:

  ```
  rlwinm  rd, ra, 32 - n, n, 31
  ```

  Rotating left by 32 – *n* is the same as rotating right by *n*. We then clear the leftmost *n* bits by saying we want to keep the bits starting from position *n* to the end of the register.

- Pluck bit *n* out of a 32-bit value:

  ```
  rlwinm  rd, ra, n + 1, 31, 31
  ```

  Rotate left by *n* + 1 to position the desired bit in position 31, then clear all the other bits.

- Extract a bitfield of length *m* at position *n*:

  ```
  rlwinm  rd, ra, n + m, 32 - m, 31
  ```

  Rotate left by *n* + *m*, which right-aligns the field, then clear all but the rightmost *m* bits.

- Take the least significant *m* bits of a value and position it so it can be inserted into a bitfield starting at position *n*:

  ```
  rlwinm  rd, ra, 32 - n - m, n, n + m - 1
  ```

  Rotate right by *n* + *m*, which positions the field into its final location, then clear all the bits that aren't used to represent the field.

- Set a bitfield of length *m* at position *n* to zero:

  ```
  rlwinm  rd, ra, 0, n + m, n - 1
  ```

  A rotation of zero does nothing, but here's where we exploit the wraparound behavior of the mask: We keep the bits starting at *n* + *m*, which is the last bit past the end of the field, and continue keeping bits through the end of the register, and then wrap around and keep bits starting at bit zero, and stop at bit *n* – 1, which is the last bit before the start of the field.

- Zero-extend a byte to a word and do not update *cr0*:

  ```
  rlwinm  rd, ra, 0, 24, 31
  ```

  We perform no rotation and zero out the most significant 24 bits. We can also do zero extension with `andi.`, but the `rlwinm` instruction lets us do it without updating *cr0*.

- Zero-extend a halfword to a word and do not update *cr0*:

  ```
  rlwinm  rd, ra, 0, 16, 31
  ```

  This time, after doing no rotation, we zero out the most significant 16 bits. Again, we could have done this with `andi.`, but this way lets us do it without updating *cr0*.

There is also a version of the instruction where the rotation amount comes from a register.

```
rlwnm   rd, ra, rb, imm5a, imm5b
rlwnm.  rd, ra, rb, imm5a, imm5b ; also updates cr0
```

"Rotate left word and mask" is like "rotate left word immediate and mask", except that the rotation amount is specified by the value of a register. (Since rotation by a multiple of 32 bits is a nop, it doesn't matter whether bits 0 through 26 in *rb* are respected or ignored.)

```
rlwimi  rd, ra, imm5a, imm5b, imm5c
rlwimi. rd, ra, imm5a, imm5b, imm5c ; also updates cr0
```

"Rotate left word immediate and mask insert" rotates the value in the *ra* register left by *imm5a* bits, and then copies bits *imm5b* through *imm5c* of the rotated value (wrapping around if necessary) to *rd*, leaving the other bits of *rd* alone. This instruction is most useful for storing a value into a bitfield:

```
rlwimi  rd, ra, 32 - n - m, n, n + m - 1
```

The above instruction takes the least significant *m* bits of *ra* and sets them into a bitfield of size *m* starting at position *n* in *rd*. We did the math for this before, when we tried out `rlwinm` to position a bitfield. By using `rlwimi`, we get to store it.

Okay, now we can get to the true shift instructions.

```
slw     rd, ra, rb      ; rd = ra << (rb % 64)
slw.    rd, ra, rb      ; rd = ra << (rb % 64), update cr0

srw     rd, ra, rb      ; rd = ra >> (rb % 64)
srw.    rd, ra, rb      ; rd = ra >> (rb % 64), update cr0
```

The `slw` and `srw` instructions shift a register left or right by an amount specified by the value of another register. Notice that the shift amount is taken mod 64, rather than mod 32. This means that a shift by 63 will set the result to zero, but a shift by 64 will do nothing.

```
sraw    rd, ra, rb      ; rd = (int32_t)ra >> (rb % 64), update carry
sraw.   rd, ra, rb      ; rd = (int32_t)ra >> (rb % 64), update carry and cr0

srawi   rd, ra, imm5    ; rd = (int32_t)ra >> imm5, update carry
srawi.  rd, ra, imm5    ; rd = (int32_t)ra >> imm5, update carry and cr0
```

The `sraw` instruction performs an arithmetic right shift by an amount specified by the *rb* register, and the `srawi` instruction does the same, but with an immediate shift amount.

These four shift instructions are special because they always update carry: The carry bit is set if and only if the original value was negative and any bits shifted out were nonzero. This rule for the carry bit allows you to follow the right-shift instruction with an `addze` to perform a division by a power of two that rounds toward zero. (If you omit the `addze`, then the right shift performs a division by a power of two that rounds toward minus infinity.)

Exercise: How would you do a division by a power of two that rounds toward positive infinity?

Here's a sample code sequence to perform a C-style logical not operation:

```
cmpwi   r3,0                ; set EQ if value was zero
mfcr    r3                  ; r3 = cr
rlwinm  r3,r3,eq+1,31,31    ; save the EQ bit
```

Similarly, you can calculate the C logical `<` and `>` operations by performing the comparison and extracting the *lt* or *gt* bit. To get the other three comparisons `!=`, `<=` and `>=`, you follow up with `xori r3, r3, 1`.

Okay, those are the logical and shifting instructions. Next time, we'll look at memory access.

Raymond Chen

Follow