The PowerPC 600 series, part 4: Bitwise operations and constants

devblogs.microsoft.com/oldnewthing/20180809-00

August 9, 2018

Raymond Chen

The PowerPC 600 series includes the following bitwise logical operations:

```
and     rd, ra, rb      ; rd =   ra &  rb
or      rd, ra, rb      ; rd =   ra |  rb
xor     rd, ra, rb      ; rd =   ra ^  rb
nand    rd, ra, rb      ; rd = ~(ra &  rb)
nor     rd, ra, rb      ; rd = ~(ra |  rb)
eqv     rd, ra, rb      ; rd = ~(ra ^  rb)
andc    rd, ra, rb      ; rd =   ra & ~rb "and complement"
orc     rd, ra, rb      ; rd =   ra | ~rb "or complement"
; also "." versions
```

Each of these instructions also comes with a dot variant that updates *cr0* based on the result.

There are also versions that take immediates or sometimes shifted immediates, and sometimes they update flags, and sometimes they don't. There isn't much orthogonality here. It's all case-by-case.

```
andi.   rd, ra, imm16   ; rd =   ra &  (uint16_t)imm16, update cr0
andis.  rd, ra, imm16   ; rd =   ra & ((uint16_t)imm16 << 16), update cr0
ori     rd, ra, imm16   ; rd =   ra |  (uint16_t)imm16
oris    rd, ra, imm16   ; rd =   ra | ((uint16_t)imm16 << 16)
xori    rd, ra, imm16   ; rd =   ra ^  (uint16_t)imm16
xoris   rd, ra, imm16   ; rd =   ra ^ ((uint16_t)imm16 << 16)
```

Immediates are allowed only on three of the bitwise operations, and the `and` version always updates flags, whereas the `or` and `xor` versions never update flags.

For some reason, sign extension is placed in the logical operations group.

```
extsb   rd, ra          ; rd = (int8_t)ra
extsb.  rd, ra          ; rd = (int8_t)ra, update cr0
extsh   rd, ra          ; rd = (int16_t)ra
extsh.  rd, ra          ; rd = (int16_t)ra, update cr0
```

We now have enough instructions to load constants.

If the constant is in the range `0xFFFF8000` to `0x00007FFF`, it can be loaded in one instruction:

```
; load immediate: rd = (int16_t)imm16
addi    rd, 0, imm16     ; li   rd, imm16
```

It can also be done in one instruction if the constant is an exact multiple of 65536.

```
; load immediate shifted: rd = imm16 << 16
addis   rd, 0, imm16     ; lis  rd, imm16
```

These take advantage of the fact that the `addi` and `addis` instructions treat *r0* as if it were zero. They are the only non-memory instructions that have this special behavior with respect to *r0*.

If the constant you want to load doesn't fall into either of the two categories above, then you'll have to load it in two steps:

```
addis   rd, 0, imm16a    ; rd =  imm16a << 16
ori     rd, rd, imm16b   ; rd = (imm16a << 16) | (uint16_t)imm16b
```

This sequence takes advantage of the fact that the `ori` instruction treats its 16-bit immediate as an unsigned value. That way, we don't have to play funny games with the most significant 16 bits if the least-significant 16 bits happen to form a negative integer when interpreted as a signed 16-bit value.

While I'm here I may as well mention a third synthetic instruction based on `addi`:

```
; load address: rd = effective address of imm16(ra)
addi    rd, ra, imm16    ; la   rd, imm16(ra)
```

A commonly-used synthetic instruction is "move register":

```
or      rd, ra, ra       ; mr  rd, ra
or.     rd, ra, ra       ; mr. rd, ra
```

Moving a register to itself is functionally a nop, but the processor overloads it to signal information about priority.

```
or      r1, r1, r1       ; low priority
or      r6, r6, r6       ; medium-low priority
or      r2, r2, r2       ; normal priority
```

A program can voluntarily set itself to low priority if it is waiting for a spin lock. There are other priority levels which are available only to kernel mode and are ignored in user mode.

Finally, everybody's favorite instruction:

```
ori     r0, r0, 0        ; nop
```

This is the official `nop` instruction recognized by the processor. There are other instructions that have no visible effect, but they might not be optimized efficiently. For example, `rlwinm ra, ra, 0, 0, 31` has no visible effect, but it will probably introduce a register dependency. And as we saw above, sometimes instructions with no visible effect become overloaded as signals to the processor, so your best bet is to avoid them.

Wait, you don't know what the `rlwinm` instruction does? We'll dig into that next time, when we enter the crazy world of rotating and shifting, and you'll be formally introduced to the `rlwinm` instruction, the Swiss army knife instruction of the PowerPC instruction set.

Raymond Chen

Follow