The PowerPC 600 series, part 6: Memory access

devblogs.microsoft.com/oldnewthing/20180813-00

August 13, 2018

Raymond Chen

The PowerPC 600 series has two addressing modes. We will demonstrate them with the `lwz` instruction, which loads a word from memory.

```
    lwz     rd, disp16(ra/0) ; load word from memory at ra/0 + (int16_t)disp16
    lwzx    rd, ra/0, rb     ; load word from memory at ra/0 + rb
```

The regular load instruction fetches a word from a memory location specified by a register and a signed 16-bit displacement. By convention, if no displacement is given, it is assumed to be zero. The Windows disassembler displays the displacement in hex without a `0x` prefix, but I'm going to put the prefix in to minimize confusion.

The indexed load instruction adds two registers to determine the address from which to load the word. Note that you cannot use *r0* as the base register for the load; if you try to use it, it comes out as zero.[1]

Both of the instructions can be suffixed with `u` ("update") to set the *ra* register equal to the effective address of the load. (If an exception occurs on the memory access, the *ra* register is not updated. This allows the instruction to be restarted.)

```
    lwzu    rd, disp16(ra)   ; load word from memory at ra + (int16_t)disp16
                             ; and then set ra equal to ra + (int16_t)disp16
                             ; ra may not be r0 or rd


    lwzxu   rd, ra, rb       ; load word from memory at ra + rb
                             ; and then set ra equal to ra + rb
                             ; ra may not be r0 or rd
```

The *ra* register cannot be *r0* because *r0* acts like the zero register during effective address calculations, and it would make no sense to update the zero register. The *ra* register cannot be the same register as *rd* because that would create a conflict between the two output registers.

The `lwzu` instruction is handy if you're walking through an array, since it lets you step to the next item and fetch a word from it in a single instruction.

Okay, so here are the ways you can load data from memory. I will present only the basic form of the instruction, but understand that `x`, `u`, and `xu` forms are also available.

```
    lbz     rd, disp16(ra/0) ; load byte and zero extend
    lhz     rd, disp16(ra/0) ; load halfword and zero extend
    lwz     rd, disp16(ra/0) ; load word and zero extend

    lha     rd, disp16(ra/0) ; load halfword and sign extend
                             ; (a = "arithmetic")
```

There is a bonus sign-extending load of halfwords, but sadly no sign-extending load of bytes.

Why does the `lwz` instruction say "and zero extend" even though there's nowhere to extend to? Because there would be a place to extend to if running on a 64-bit version of the processor. (Windows NT runs the processor in 32-bit mode, but the 64-bit registers are available if the processor supports them.)

There is a corresponding set of store instructions.

```
    stb     rd, disp16(ra/0) ; store byte
    sth     rd, disp16(ra/0) ; store halfword
    stw     rd, disp16(ra/0) ; store word
    ; also "x", "u", and "xu" variants.
```

In particular, the `stwu` instruction is extremely handy when setting up your stack frame, which we'll see later when we learn about software conventions.

All loads and stores should be to suitably-aligned locations. The architecture permits but does not require the processor to support unaligned memory access in little-endian mode, and even if it does support unaligned loads, it might do so only partially. (For example, it might support unaligned loads provided they do not span multiple cache lines.) As noted earlier, if an unaligned store crosses into an invalid page, the processor is permitted to store the valid part before the exception is raised. If the processor does not support an unaligned operation, it will trap, and the kernel will emulate it.

There are no special instructions for assisting with unaligned loads. You're on your own:

```
; load halfword unaligned from n(r3) into r4 with zero extension
; requires a scratch register r5.
lbz     r4, n(r3)         ; r4 = least significant byte
lbz     r5, n+1(r3)       ; r5 = most significant bytes
rlwimi  r4, r5, 8, 0, 23  ; merge together

; load halfword unaligned from n(r3) into r4 with sign extension
; requires a scratch register r5.
lbz     r4, n(r3)         ; r4 = least significant byte
lba     r5, n+1(r3)       ; r5 = most significant bytes (sign extended)
rlwimi  r4, r5, 8, 0, 23  ; merge together

; load word unaligned from n(r3) into r4
; requires a scratch register r5.
lbz     r4, n(r3)         ; r4 = least significant byte
lbz     r5, n+1(r3)       ; r5 = next most significant byte
rlwimi  r4, r5, 8, 16, 23 ; merge together
lbz     r5, n+2(r3)       ; r5 = next most significant byte
rlwimi  r4, r5, 16, 8, 15 ; merge together
lbz     r5, n+3(r3)       ; r5 = most significant byte
rlwimi  r4, r5, 24, 0, 7  ; merge together
```

To load an unaligned value, you load up the individual bytes and merge them using `rlwimi`.

```
; store halfword unaligned from r4 to n(r3)
stb     r4, n(r3)         ; store least significant byte
rlwinm  r4, r4, 24, 0, 31 ; rotate right 8 bits
stb     r4, n+1(r3)       ; store next significant byte
rlwinm  r4, r4, 8, 0, 31  ; rotate back to original value
                          ; (in case you still need the value)

; store word unaligned from r4 to n(r3)
stb     r4, n(r3)         ; store least significant byte
rlwinm  r4, r4, 24, 0, 31 ; rotate right 8 bits
stb     r4, n+1(r3)       ; store next significant byte
rlwinm  r4, r4, 24, 0, 31 ; rotate right 8 bits
stb     r4, n+2(r3)       ; store next significant byte
rlwinm  r4, r4, 24, 0, 31 ; rotate right 8 bits
stb     r4, n+3(r3)       ; store next significant byte
rlwinm  r4, r4, 24, 0, 31 ; rotate back to original value
                          ; (in case you still need the value)
```

To store an unaligned value, you store the individual bytes. Since the `stb` instruction stores the last significant byte, each byte of value takes its turn in the least significant position. In practice, you are more likely to see the compiler extract the bytes into a separate register to avoid long dependency chains, at the cost of an additional register.

```
    ; store halfword unaligned from r4 to n(r3), using r5 as scratch
    stb     r4, n(r3)         ; store least significant byte
    rlwinm  r5, r4, 24, 0, 31 ; extract next significant byte
    stb     r5, n+1(r3)       ; store next significant byte

    ; store word unaligned from r4 to n(r3), using r5 as scratch
    stb     r4, n(r3)         ; store least significant byte
    rlwinm  r5, r4, 24, 0, 31 ; extract next significant byte
    stb     r5, n+1(r3)       ; store next significant byte
    rlwinm  r5, r4, 16, 0, 31 ; extract next significant byte
    stb     r5, n+2(r3)       ; store next significant byte
    rlwinm  r5, r4,  8, 0, 31 ; extract next significant byte
    stb     r5, n+3(r3)       ; store next significant byte
```

Okay, back to addressing modes: Treating *r0* as zero for effective address computations gives you absolute addressing to the lowest and highest 32KB of memory. This isn't particularly useful in Windows NT, but I can see how it would be handy in an embedded system where there is no virtual memory. You could map the ROM to the low 32KB and RAM to the high 32KB, and now you have absolute addressing to your entire system.

If you need absolute addressing to anything outside the top and bottom 32KB of address space, you'll have to do something else. One way is to build up the address as a 32-bit constant, like we saw earlier. But the PowerPC takes a different approach: By convention, the *r2* register contains a value called the *table of contents*. But there are some other topics I want to get through before I dig into the Windows NT software conventions, so you'll have to be a bit patient.

Bonus chatter: There are additional instructions available in big-endian mode for loading and storing multiple registers, but they are not available in little-endian mode, so I won't cover them.

[1] Though if you really wanted to perform a load from *r0*, I guess you could use the indexed load

```
    lwzx    rd, 0, r0       ; load word from memory at 0 + r0
```

Raymond Chen

Follow