The SuperH-3, part 8: Bit shifting

devblogs.microsoft.com/oldnewthing/20190814-00

August 14, 2019

Raymond Chen

The bit shifting operations are fairly straightforward.

```
; arithmetic (signed) shifts
SHAL Rn         ; Rn <<= 1, T = the bit shifted out
SHAR Rn         ; Rn >>= 1, T = the bit shifted out

; logical (unsigned) shifts
SHLL Rn         ; Rn <<= 1, T = the bit shifted out
SHLR Rn         ; Rn >>= 1, T = the bit shifted out
SHLL2 Rn        ; Rn <<= 2
SHLR2 Rn        ; Rn >>= 2
SHLL8 Rn        ; Rn <<= 8
SHLR8 Rn        ; Rn >>= 8
SHLL16 Rn       ; Rn <<= 16
SHLR16 Rn       ; Rn >>= 16
```

You cannot shift by arbitrary constant amounts. Only certain fixed values are permitted. If you want to shift left by, say, 9, you'll have to construct it from a `SHLL8` and a `SHLL`.

Note also that `SHAL` and `SHLL` are functionally equivalent. But they have different encodings, so the designers burned an opcode for a redundant operation.

There are no "large shift" options for right shifts. You can perform multiple one-bit shifts, or use a variable shift:

```
SHAD Rm, Rn     ; if Rm > 0: Rn <<= (31 & Rm)
                ; if Rm = 0: nop
                ; if Rm < 0: Rn >>= (31 & -Rm), signed

SHLD Rm, Rn     ; if Rm > 0: Rn <<= (31 & Rm)
                ; if Rm = 0: nop
                ; if Rm < 0: Rn >>= (31 & -Rm), unsigned
```

Note that these shift instructions shift both left *and* right, depending on the sign of the shift amount. If you want to shift right by an amount in a register, you therefore need to negate the value, and then shift left.

Finally, we have rotation.

```
ROTL Rn         ; rotate left, T contains carried-out bit
ROTR Rn         ; rotate right, T contains carried-out bit
ROTCL Rn        ; 33-bit rotate through T
ROTCR Rn        ; 33-bit rotate through T
```

The rotation instructions rotate either a 32-bit or 33-bit value by one position. For the 32-bit rotations, the bit that rotated off the end is copied to *T*. For the 33-bit rotations, the *T* flag acts as the 33rd bit.

We saw earlier that there is no `NEGV` instruction. To detect overflow from a negation, you just have to check for the value `0x80000000` directly. Here's the shortest sequence I could come up with:

```
; branch if Rn equals 0x80000000
rotl Rn         ; rotate left one bit
dt   Rn         ; decrement and test for zero
bt   underflow ; Y: underflow occurred
```

The result of the `DT` is zero if the previous value was 1, and the previous value was 1 if the original value was `0x80000000`.

This is a destructive operation, so do it in a scratch register. You should have one available, since it's the source register for the `NEGV` you were checking.

We'll look more at constants next time.

Raymond Chen

Follow