The SuperH-3, part 5: Multiplication

devblogs.microsoft.com/oldnewthing/20190809-00

August 9, 2019

Raymond Chen

Last time, we looked at simple addition and subtraction. Now let's look at multiplication.

Multiplication operations report their results in a pair of 32-bit registers called called *MACH* and *MACL*, which collectively form a 64-bit virtual register known as *MAC* (multiply and accumulate).

We start with the simple multiplication operations.

```
MUL.L   Rm, Rn  ; MACL =           Rm *            Rn, no effect on MACH
MULS.W  Rm, Rn  ; MACL = ( int16_t)Rm * ( int16_t)Rn, no effect on MACH
MULU.W  Rm, Rn  ; MACL = (uint16_t)Rm * (uint16_t)Rn, no effect on MACH
```

The `.W` operations treat the two source operands as 16-bit values, either signed or unsigned, and store the 32-bit result into *MACL*. The `MUL.L` treats the source operands as full 32-bit values, and produces a 32-bit result in *MACL*. (It doesn't matter whether the sources are considered signed or unsigned because the lower 32 bits of the result are the same either way.)

The next instructions produce 64-bit results.

```
DMULS.L Rm, Rn      ; MAC = Rn * Rm,   signed 32x32→64 multiply
DMULU.L Rm, Rn      ; MAC = Rn * Rm, unsigned 32x32→64 multiply

MAC.L   @Rm+, @Rn+  ; MAC += @Rm++ * @Rn++, signed 32x32→64 multiply
MAC.W   @Rm+, @Rn+  ; MAC += @Rm++ * @Rn++, signed 16x16→64 multiply
```

The `MAC.x` instructions are interesting in that they access two memory locations in one instruction. Both *Rm* and *Rn* are treated as addresses, 16-bit or 32-bit values are loaded from those addresses, the loaded values are treated as signed integers, multiplied together, and the result added to the 64-bit accumulator register *MAC*, and finally the registers are incremented by the operand size. The design of the instruction is evidently for performing a dot product of two vectors.

There's an additional wrinkle to the `MAC.x` instructions: If you set the *S* flag, then the operations use saturating addition rather than wraparound addition. For `MAC.L`, the saturation is as a 48-bit value, and the value is sign-extended to a 64-bit value in *MAC*. For `MAC.W`, the saturation is as a 32-bit value, and the bottom bit of *MACH* is set to 1 if an overflow occurred.

In practice, of these multiplication instructions, you will likely see only `MUL.L` in compiler-generated code.

Oh wait, how do you get the answers out of the *MAC* registers? Yeah, there are instructions for that too.

```
    CLRMAC                ; MAC = 0

    LDS      Rm, MACH     ; MACH = Rm
    LDS      Rm, MACL     ; MACL = Rm
    LDS.L    @Rm+, MACH   ; MACH = @Rm+
    LDS.L    @Rm+, MACL   ; MACL = @Rm+

    STS      MACH, Rn     ; Rn = MACH
    STS      MACL, Rn     ; Rn = MACL
    STS.L    MACH, @-Rn   ; @-Rn = MACH
    STS.L    MACL, @-Rn   ; @-Rn = MACL
```

The `CLRMAC` instruction sets *MAC* to zero, which is a good starting point for subsequent `MAC.x` instructions.

The `LDS` instructions move values into the *MAC* registers. You can move a value directly from a register or load it (with post-increment) from memory. Conversely, the `STS` instructions move values out of the *MAC* registers, either into a general-purpose register or into memory.

Next up is integer division, which is going to be interesting.

Raymond Chen

Follow