The SuperH-3, part 13: Misaligned data, and converting between signed vs unsigned values

August 21, 2019

Raymond Chen

When going through compiler-generated assembly language, there are some patterns you'll see over and over again. Note that the code you see may not look exactly like this due to compiler instruction scheduling. In particular, the sequences for misaligned memory access may bring additional registers into play in order to avoid register dependencies.

First, is the unsigned memory access. Bytes and words loaded from memory are sign-extended by default. If you want to load an unsigned value, you need to perform an explicit zero-extension.

```
; load unsigned byte from address in r0
MOV.B   @r0, r1         ; loads sign-extended byte
EXTU.B  r1, r1          ; zero-extend the byte to a longword

; load unsigned word from address in r0
MOV.W   @r0, r1         ; loads sign-extended word
EXTU.W  r1, r1          ; zero-extend the word to a longword
```

Next up is misaligned data. The SH-3 does not support unaligned memory access. Not only that, but the kernel doesn't even emulate unaligned memory access. If you access memory from a misaligned address, you take an access violation and your process crashes. So don't mess up!

There are no special instructions for accessing misaligned data. You are on your own to take individual bytes and combine them into the desired final value, or to take the starting value and decompose it into bytes.

```
; store 16-bit value in r1 to possibly unaligned address in r0
; destroys r1
;                            r1      @r0
;                          xxxxAABB  xx xx
MOV.B   r1, @r0          ; xxxxAABB  BB xx
SHLR8   r1               ; 00xxxxAA  BB xx
MOV.B   r1, @(1, r0)     ; 00xxxxAA  BB AA

; store 32-bit value in r1 to possibly unaligned address in r0
; destroys r1
;                            r1      @r0
;                          AABBCCDD  xx xx xx xx
MOV.B   r1, @r0          ; AABBCCDD  DD xx xx xx
SHLR8   r1               ; 00AABBCC  DD xx xx xx
MOV.B   r1, @(1, r0)     ; 00AABBCC  DD CC xx xx
SHLR8   r1               ; 0000AABB  DD CC xx xx
MOV.B   r5, @(2, r0)     ; 0000AABB  DD CC BB xx
SHLR8   r1               ; 000000AA  DD CC BB xx
MOV.B   r1, @(3, r0)     ; 000000AA  DD CC BB AA

; read 16-bit value from possibly unaligned address in r0
;                            r1      r2        @r0
;                          xxxxxxxx  xxxxxxxx  BB AA
MOV.B   @(1, r0), r1     ; SSSSSSAA  xxxxxxxx
SHLL8   r1               ; SSSSAA00  xxxxxxxx
MOV.B   @r0, r2          ; SSSSAA00  SSSSSSBB
EXTU.B  r2, r2           ; SSSSAA00  000000BB
OR      r1, r2           ; SSSSAA00  SSSSAABB
                         ; r2 contains signed 16-bit value
EXTU.W  r2, r2           ; SSSSAA00  0000AABB
                         ; r2 contains unsigned 16-bit value

; read 32-bit value from possibly unaligned address in r0
;                            r1      r2        @r0
;                          xxxxxxxx  xxxxxxxx  DD CC BB AA
MOV.B   @(3, r0), r1     ; SSSSSSAA  xxxxxxxx
SHLL8   r1               ; SSSSAA00  xxxxxxxx
MOV.B   @(2, r0), r2     ; SSSSAA00  SSSSSSBB
EXTU.B  r2, r2           ; SSSSAA00  000000BB
OR      r2, r1           ; SSSSAABB  000000BB
SHLL8   r1               ; SSAABB00  000000BB
MOV.B   @(1, r0), r2     ; SSAABB00  SSSSSSCC
EXTU.B  r2, r2           ; SSAABB00  000000CC
OR      r2, r1           ; SSAABBCC  000000CC
SHLL8   r1               ; AABBCC00  000000CC
MOV.B   @r0, r2          ; AABBCC00  SSSSSSDD
EXTU.B  r2, r2           ; AABBCC00  000000DD
OR      r1, r2           ; AABBCC00  AABBCCDD
```

Less often, you will see code that sign-extends a 32-bit value to a 64-bit value.

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
MOV     r0, r1          ; copy value to r1
SHLL    r1              ; T contains high bit of value
SUBC    r1, r1          ; if T=0, then r1 = 00000000
                        ; if T=1, then r1 = FFFFFFFF
```

If you happen to have the value 0 lying around in a register, you could accomplish the task in two instructions:

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
; assumes that r2 already contains the value zero
CMP/GT  r0, r2          ; T = (0 > r0)
                        ; in other words, T=0 if r0 is positive or zero
                        ;                 T=1 if r0 is negative
SUBC    r1, r1          ; if T=0, then r1 = 00000000
                        ; if T=1, then r1 = FFFFFFFF
```

That is just code golf on my part. I haven't seen the compiler use this trick, or the next one.

```
; sign-extend 32-bit value in r0 to 64-bit value in r1:r0
; preserves flags
ROTCL   r0              ; rotate r0 left, copying high bit into T
                        ; and saving old T in low bit of r0
SUBC    r1, r1          ; if T=0, then r1 = 00000000, T stays 0
                        ; if T=1, then r1 = FFFFFFFF, T stays 1
ROTCR   r0              ; rotate r0 right to restore original value
                        ; and recover original value of T
```

In general, you'll see that SH-3 assembly code is somewhat verbose, even more so because compiler technology back in this time period was not as advanced as it is today, but you have to realize that each of these instructions is only half the size of the instructions of its RISC-style contemporaries, so even though you plowed through 2000 instructions, that's only 4KB of code.

Okay, next time, we're returning to reality and looking at function call patterns.

Raymond Chen

Follow