The ARM processor (Thumb-2), part 9: Sign and zero extension

devblogs.microsoft.com/oldnewthing/20210610-00

June 10, 2021

Raymond Chen

I noted last time that you could use the bitfield extraction instructions to do zero- and sign-extension of bytes and halfwords to words. But there are dedicated instructions for these operations which have smaller encodings if the source and destination registers are low.

```
; unsigned extend byte to word
uxtb    Rd, Rm      ; Rd = (uint8_t)Rm

; signed extend byte to word
sxtb    Rd, Rm      ; Rd = (int8_t)Rm

; unsigned extend halfword to word
uxth    Rd, Rm      ; Rd = (uint16_t)Rm

; signed extend halfword to word
sxth    Rd, Rm      ; Rd = (int16_t)Rm
```

You can optionally apply a rotation to the second register so that you can extract a 8-bit or 16-bit value that sits along a byte boundary.

```
; unsigned/signed extend byte to word with rotation
; rotation must be a multiple of 8
uxtb    Rd, Rm, #rot ; Rd = (uint8_t)(Rm ROR #rot)
sxtb    Rd, Rm, #rot ; Rd = ( int8_t)(Rm ROR #rot)

; unsigned/signed extend halfword to word with rotation
; rotation must be a multiple of 8
uxth    Rd, Rm, #rot ; Rd = (uint16_t)(Rm ROR #rot)
sxth    Rd, Rm, #rot ; Rd = ( int16_t)(Rm ROR #rot)
```

It's kind of weird to apply a 24-bit rotation to extract a halfword, but you can do it if you want to.

You can also zero-extend or sign-extend a word to a doubleword using instructions you already have available:

```
; zero-extend Rd to Rd/R(d+1)
mov     R(d+1), #0          ; set to 0

; sign-extend Rd to Rd/R(d+1)
asrs    R(d+1), Rd, #31     ; copy sign bit to all bits
```

The trick is that a signed right-shift by 31 positions ends up filling the entire word with the sign bit. We use the S-version `ASRS` because it allows a compact 16-bit encoding if both the source and destination registers are low.

The `ASR #31` trick can also be used in the `op2` of arithmetic or logical instructions.

```
; set r0 to zero if r1 is positive or zero
and     r0, r1, ASR #31
```

The trick here is that `r1, ASR #31` produces `0xFFFFFFFF` if *r1* is negative, but `0x00000000` if *r1* is positive or zero.

In addition to the straight zero- and sign-extension operations, there are other instructions that combine the extension with another operation. Most of them are focused on multimedia scenarios, but the extend-and-add instructions are more general-purpose, and I have seen the compiler generate the versions with no rotation.

```
; zero/sign extend and add byte with optional rotation
; rotation must be a multiple of 8
uxtab   Rd, Rn, #rot        ; Rd = Rd + (uint8_t)(Rn ROR #rot)
sxtab   Rd, Rn, #rot        ; Rd = Rd + ( int8_t)(Rn ROR #rot)

; zero/sign extend and add halfword with optional rotation
; rotation must be a multiple of 8
sxtah   Rd, Rn, #rot        ; Rd = Rd + ( int16_t)(Rn ROR #rot)
uxtah   Rd, Rn, #rot        ; Rd = Rd + (uint16_t)(Rn ROR #rot)
```

There's another instruction that looks like it'd come in handy, particularly in Win32 user interface code that has to pack two 16-bit coordinates into a 32-bit integer, but I haven't seen any compiler generate it:

```
; pack halfword bottom-and-top, or top-and-bottom
; shift is optional
pkhbt   Rd, Rn, Rm, LSL #imm ; Rd = ((Rm LSL #imm) << 16) | (uint16_t)Rn
pkhtb   Rd, Rn, Rm, ASR #imm ; Rd = (Rn << 16) | (uint16_t)(Rm ASR #imm)
```

The bottom-and-top version puts the first input register in the bottom part of the output, and the second input parameter goes into the top part. The top-and-bottom version does it the other way. (The top-and-bottom instruction is not redundant because the barrel shifter can be applied only to the second input parameter.)

When the compiler needs to do this, it generates two instructions:

```
; pack halfword bottom-and-top
uxth    r12, Rn                 ; r12 = (uint16_t)Rn
orr     Rd, r12, Rm, LSL #16    ; Rd = r12 | (Rm << 16)
                                ;    = (uint16_t)Rn | (Rm << 16)
```

Even if it didn't want to use `PKHBT`, it could have used `BFI` to pack the values in a single instruction:

```
; pack halfword bottom-and-top (in place)
bfi     Rd, Rm, #16, #16        ; Rd[31:16] = Rm[15:0]
```

Maybe there's some dirty secret about the `PKHBT` and `BFI` instructions that the compiler knows but I don't.

Raymond Chen

Follow