Division

Pieter P

NEON doesn't have any integer division instructions, because they are expensive to implement in hardware. Luckily, you can often replace divisions by other instructions, like bit shifts and multiplications.

Dividing by powers of two

Unsigned integers

Flooring

To divide an unsigned integer by 2^k, you can just use a bit shift by k bits to the right.

```
uint16_t div_by_16_a(uint16_t x) {
    return x / 16;
}

uint16_t div_by_16_b(uint16_t x) {
    return x >> 4; // 16 = 2⁴
}
```

Of course, any half-decent compiler will produce the same instructions for both of the functions above, so it's much better to write `x / 16`, because it clearly shows your intent.

To write the same division for NEON, you can use the `vshr_n_u16` intrinsic:

```
#include <arm_neon.h>

uint16x4_t div_by_16(uint16x4_t x) {
    return vshr_n_u16(x, 4); // 16 = 2⁴
}

uint16x8_t div_by_16(uint16x8_t x) {
    return vshrq_n_u16(x, 4); // 16 = 2⁴
}
```

The name is derived from vector shift right, `n` indicates a fixed number of bits is used, and `u16` is the type of elements in the vector (16-bit unsigned integers in this example).
There are two versions, the one without the `q` suffix operates on double-word vector registers (2×32 bits), and the one with the `q` suffix operates on quad-word vector registers (4×32 bits).

Rounding

NEON also has instructions for rounding right shifts, for example, `vrshr_n_u16` (note the `r` prefix):

```
uint16x8_t div_by_16_round(uint16x8_t x) {
    return vrshrq_n_u16(x, 4); // 16 = 2⁴
}
```

Signed integers

Flooring

Dividing signed integers by a power of two is a little bit trickier than dividing unsigned integers, because you cannot simply use a bit shift. Consider $-7/16$ as an example: you would expect the result to be 0. However, $-7 \gg 4$ turns out to be -1.

To get around this problem, you have to add the divisor minus one if the dividend is negative:

```
int16_t div_by_16_a(int16_t x) {
    return x / 16;
}

int16_t div_by_16_b(int16_t x) {
    if (x < 0)
        x += 16 - 1;
    return x >> 4; // 16 = 2⁴
}
```

NEON doesn't have any conditional instructions, but it does have compare instructions that can be used to generate a mask. The compare instructions return either all zeros (`0x0000`) or all ones (`0xFFFF`), so by using a bitwise AND, they can be used to conditionally enable or disable the correction term.

```
int16x8_t div_by_16(int16x8_t x) {
    int16x8_t negative = vcltzq_s16(x); // compare less than zero
    int16x8_t correction = vdupq_n_s16(16 - 1);
    correction = vandq_s16(negative, correction); // only add correction if < 0
    x = vaddq_s16(x, correction);
    return vshrq_n_s16(x, 4); // 16 = 2⁴
}
```

In this snippet, `vdupq_n_s16` is used to duplicate a single value across all lanes of a vector register. Sadly, we cannot use the correction $(16 - 1)$ as an immediate argument to the AND instruction, because you can only set one of the bytes of the immediate value at a time. The other byte is always implicitly `0xFF`. In other words, the only possible immediate operands to AND for 16-bit numbers are `0xFFXY` and `0xXYFF`.
$16 - 1 =$ `0x000F` doesn't fit this pattern. However, $1 - 16 =$ `0xFFF1` does. By replacing the addition of 15 by a subtraction of -15, we can save one move instruction (`vdupq_n_s16`), and use an immediate AND instruction instead.

```
#define vandq_n_u16(a, b)                                                   \
    __extension__({                                                         \
        uint16x8_t a_ = (a);                                                \
        uint16_t b_ = ~(b);                                                 \
        __asm__("bic %0.8h, #%1" : "+w"(a_) : "i"(b_) : /* No clobbers */); \
        a_;                                                                 \
    })

int16x8_t div_by_16(int16x8_t x) {
    uint16x8_t negative = vcltzq_s16(x); // compare less than zero
    uint16x8_t correction = vandq_n_u16(negative, 1 - 16);
    x = vsubq_s16(x, vreinterpretq_s16_u16(correction));
    return vshrq_n_s16(x, 4); // 16 = 2⁴
}
```

There is no actual AND instruction, the BIC (bit clear) instruction is used instead. There's no intrinsic for it, so we have to add a line of inline assembly. A compiler extension is used, because parameter `b` should be a compile-time constant, it is used to compute the immediate field of the instruction, so it cannot change at run-time.

This trick will only work for divisors less than or equal to 256, since the immediate field of the BIC instruction is only 8 bits wide. For divisors 512 and larger, the correction term will be 9 bits or more, so it doesn't fit anymore. Luckily, in that case, the low byte of the correction term will always be `0xFF`, which means that we can use the second variant of the immediate AND, where the immediate is `0xXYFF`. In practice, this is done by setting the LSL (logical shift left) amount of the BIC instruction to 8, so we can set the high byte of the immediate.

```
#define vandq_n_high_u16(a, b)                                              \
    __extension__({                                                         \
        uint16x8_t a_ = (a);                                                \
        uint16_t b_ = (~(b)) >> 8;                                          \
        __asm__("bic %0.8h, #%1, LSL #8" : "+w"(a_) : "i"(b_) :);           \
        a_;                                                                 \
    })

int16x8_t div_by_512(int16x8_t x) {
    uint16x8_t negative = vcltzq_s16(x); // compare less than zero
    uint16x8_t correction = vandq_n_high_u16(negative, 512 - 1);
    x = vaddq_s16(x, vreinterpretq_s16_u16(correction));
    return vshrq_n_s16(x, 9); // 512 = 2⁹
}
```

Rounding

For the rounding division by a power of two, we'll again use the rounding right shift. If the dividend is negative, we still need a correction. This time we just have to subtract one as a correction.

```
int16x8_t div_by_16_round(int16x8_t x) {
    uint16x8_t sign = vshrq_n_u16(vreinterpretq_u16_s16(x), 15); // sign bit
    x = vqsubq_s16(x, vreinterpretq_s16_u16(sign)); // subtract 1 if < 0
    return vrshrq_n_s16(x, 4);                      // 16 = 2⁴
}
```

Subtracting one from a negative number has an annoying edge case: if the dividend is -65536, subtracting one will wrap around to 65535, which is not what we want.
The solution is to use a saturating subtraction, such that $-65536 - 1 \mapsto -65536$. Saturating intrinsics have a `q` prefix.

To determine the sign of the dividend, we just extract the sign bit by shifting it 15 bits to the right. It's important to use an unsigned bit shift, because a signed shift would sign extend the number after shifting, resulting in `0xFFFF` instead of `0x0001`.
The sign bit is one if the number is negative, zero if it's positive or zero.

Dividing by 255

When working with images, most data is represented by 8-bit bytes, as a value from 0 to 255. For example, when alpha blending two images, you'll often need to multiply two 8-bit values together, and afterwards, you have to re-normalize 16-bit products by dividing by 255.

Approximating by a division by 256

As a first approximation, you could start by dividing by 256, because this is simply a right bit shift by 8, as discussed above. To convert from the 16-bit product back to an 8-bit number, you can use a narrowing bit shift:

```
uint8x8_t div_by_256(uint16x8_t x) {
    return vshrn_n_u16(x, 8); // 256 = 2⁸
}
```

Narrowing intrinsics have an `n` suffix. You can also realize a rounding division by using `vrshrn_n_u16`.

The main problem with this approach is that it will produce a result that's one unit too low, about 50% of the time. For example, blending white with white will result in really light gray, but not quite white ($255 \cdot 255/256 \approx 254.00 \neq 255$). In this case, rounding won't help either.
For most real-time applications, this is not an issue, and speed might be more important than perfect accuracy.

True division by 255

Flooring

The divisor of 255 can be approximated by the fraction $\texttt{0x800000}/\texttt{0x8081} = 2^{23}/\texttt{0x8081} \approx 254.996$. This means that you can replace the division by 255 by a multiplication by `0x8081`, followed by a right bit shift by 23 bits.

The downside of this approach is that a 32-bit multiplication is required. NEON doesn't have any multiplication instructions that return the high half of the product. It does have an instruction that can be used to extract all high half-words from two vector registers and merge them into a single register, as a deinterleaving or unzip operation. This is equivalent with a right bitshift of 16 bits. The instruction is `uzp2`, we'll use the `vuzp2q_u16` intrinsic.

```
uint8x8_t div_by_255(uint16x8_t x) {
    // Multiply by 0x8081 as 32-bit integers (high and low elements separately)
    // 0x800000/0x8081 ≃ 255
    uint32x4_t h = vmull_high_n_u16(x, 0x8081);
    uint32x4_t l = vmull_n_u16(vget_low_u16(x), 0x8081);
    // Extract the 16 high bits of all 32-bit products (division by 0x10000)
    x = vuzp2q_u16(vreinterpretq_u16_u32(l), vreinterpretq_u16_u32(h));
    // Divide by 0x80 and narrow from 16 bits to 8 bits
    return vshrn_n_u16(x, 7);
}
```

The effect of the `uzp2` instruction is visualized in the following diagram. Each cell is a 16-bit half-word. The two rows on the left represent the two 128-bit vector registers containing the products of the four high and low elements respectively (variables `h` and `l` in the code). Two 16-bit cells make up one of the eight 32-bit products.

H_7	L_7	H_6	L_6	H_5	L_5	H_4	L_4
H_3	L_3	H_2	L_3	H_1	L_1	H_0	L_0

UZP2
→

H_7	H_6	H_5	H_4	H_3	H_2	H_1	H_0

$H_7\ L_7$ = `x[7] * 0x8081`, where `x[7]` is the seventh element of the input, 16 bits wide. `0x8081` is the constant factor and is also 16 bits wide. Their product will be 32 bits wide, it consists of a high half-word (H_7) and a low half-word (L_7). The low half-words will be discarded by the unzip operation, effectively dividing each product by `0x10000`.

For this unzip operation to work, the vector registers have to be interpreted as 8×16-bit values (even though they actually contain 4×32-bit values).

Finally, the 16-bit high half-words of the products are narrowed to 8-bit bytes, while shifting them another 7 bits to the right for the final division by `0x80`.

The result is exact for all possible values of `x`.

Rounding

It is tempting to just replace the final shift (`vshrn_n_u16`) with a rounding shift (`vrshrn_n_u16`), in an attempt to implement a rounding division by 255. Unfortunately, this doesn't work, because rounding the final division by 128 is not the same as rounding the original division by 255. For some inputs, the output will be one unit too low. It is a pretty good approximation though, producing an incorrect result for only 127 out of all 65536 possible input values (it is correct in 99.806% of the cases, with an error of either 0 or -1).

The correct solution is to add the rounding constant *before* the multiplication. However, we're adding a rounding constant of $128 = 256/2$, while in theory, we should be using $127.5 = 255/2$. It turns out that this error of 0.5 can be compensated by changing the factor in the next step from `0x8081` to `0x8080`.

```
uint8x8_t div_by_255_round(uint16x8_t x) {
    // Add the rounding constant
    x = vaddq_u16(x, vdupq_n_u16(1 << 7));
    // Multiply by 0x8080 as 32-bit integers (high and low elements separately)
    uint32x4_t h = vmull_high_n_u16(x, 0x8080);
    uint32x4_t l = vmull_n_u16(vget_low_u16(x), 0x8080);
    // Extract the 16 high bits of all 32-bit products (division by 0x10000)
    x = vuzp2q_u16(vreinterpretq_u16_u32(l), vreinterpretq_u16_u32(h));
    // Divide by 0x80 and narrow from 16 bits to 8 bits
    return vshrn_n_u16(x, 7);
}
```

The result of this function is correct as long as the addition of the rounding constant doesn't cause overflow. This condition is satisfied if the input is less than $2^{16} - 128 = 65408$. Luckily, in this case, the result would be meaningless anyway, because the quotient would be too large to fit an 8-bit byte ($65408/255 \approx 256.5 > 255$).