On finding the average of two unsigned integers without overflow

devblogs.microsoft.com/oldnewthing/20220207-00

February 7, 2022

Raymond Chen

Finding the average of two unsigned integers, rounding toward zero, sounds easy:

```
unsigned average(unsigned a, unsigned b)
{
    return (a + b) / 2;
}
```

However, this gives the wrong answer in the face of integer overflow: For example, if unsigned integers are 32 bits wide, then it says that `average(0x80000000U, 0x80000000U)` is zero.

If you know which number is the larger number (which is often the case), then you can calculate the width and halve it:

```
unsigned average(unsigned low, unsigned high)
{
    return low + (high - low) / 2;
}
```

There's another algorithm that doesn't depend on knowing which value is larger, the U.S. patent for which expired in 2016:

```
unsigned average(unsigned a, unsigned b)
{
    return (a / 2) + (b / 2) + (a & b & 1);
}
```

The trick here is to pre-divide the values before adding. This will be too low if the original addition contained a carry from bit 0 to bit 1, which happens if bit 0 is set in both of the terms, so we detect that case and make the necessary adjustment.

And then there's the technique in the style known as SWAR, which stands for "SIMD within a register".

```
unsigned average(unsigned a, unsigned b)
{
    return (a & b) + (a ^ b) / 2;
}
```

If your compiler supports integers larger than the size of an `unsigned`, say because `unsigned` is a 32-bit value but the native register size is 64-bit, or because the compiler supports multiword arithmetic, then you can cast to the larger data type:

```
unsigned average(unsigned a, unsigned b)
{
    // Suppose "unsigned" is a 32-bit type and
    // "unsigned long long" is a 64-bit type.
    return ((unsigned long long)a + b) / 2;
}
```

The results would look something like this for processor with native 64-bit registers. (I follow the processor's natural calling convention for what is in the upper 32 bits of 64-bit registers.)

```
// x86-64: Assume ecx = a, edx = b, upper 32 bits unknown
    mov     eax, ecx        ; rax = ecx zero-extended to 64-bit value
    mov     edx, edx        ; rdx = edx zero-extended to 64-bit value
    add     rax, rdx        ; 64-bit addition: rax = rax + rdx
    shr     rax, 1          ; 64-bit shift:    rax = rax >> 1
                            ;                  result is zero-extended
                            ; Answer in eax

// AArch64 (ARM 64-bit): Assume w0 = a, w1 = b, upper 32 bits unknown
    uxtw    x0, w0          ; x0 = w0 zero-extended to 64-bit value
    add     x0, w1, uxtw    ; 64-bit addition: x0 = x0 + (uint32_t)w1
    ubfx    x0, x0, 1, 32   ; Extract bits 1 through 32 from result
                            ; (shift + zero-extend in one instruction)
                            ; Answer in x0

// Alpha AXP: Assume a0 = a, a1 = b, both in canonical form
    insll   a0, #0, a0      ; a0 = a0 zero-extended to 64-bit value
    insll   a1, #0, a1      ; a1 = a1 zero-extended to 64-bit value
    addq    a0, a1, v0      ; 64-bit addition: v0 = a0 + a1
    srl     v0, #1, v0      ; 64-bit shift:    v0 = v0 >> 1
    addl    zero, v0, v0    ; Force canonical form
                            ; Answer in v0

// MIPS64: Assume a0 = a, a1 = b, sign-extended
    dext    a0, a0, 0, 32   ; Zero-extend a0 to 64-bit value
    dext    a1, a1, 0, 32   ; Zero-extend a1 to 64-bit value
    daddu   v0, a0, a1      ; 64-bit addition: v0 = a0 + a1
    dsrl    v0, v0, #1      ; 64-bit shift:    v0 = v0 >> 1
    sll     v0, #0, v0      ; Sign-extend result
                            ; Answer in v0

// Power64: Assume r3 = a, r4 = b, zero-extended
    add     r3, r3, r4      ; 64-bit addition: r3 = r3 + r4
    rldicl  r3, r3, 63, 32  ; Extract bits 63 through 32 from result
                            ; (shift + zero-extend in one instruction)
                            ; result in r3

// Itanium Ia64: Assume r32 = a, r4 = b, upper 32 bits unknown
    extr    r32 = r32, 0, 32    // zero-extend r32 to 64-bit value
    extr    r33 = r33, 0, 32 ;; // zero-extend r33 to 64-bit value
    add.i8  r8 = r32, r33 ;;    // 64-bit addition: r8 = r32 + r33
    shr     r8 = r8, 1          // 64-bit shift:    r8 = r8 >> 1
```

Note that we must ensure that the upper 32 bits of the 64-bit registers are zero, so that any leftover values in bit 32 don't infect the sum. The instructions to zero out the upper 32 bits may be elided if you know ahead of time that they are already zero. This is common on x86-64 and AArch64 since those architectures naturally zero-extend 32-bit values to 64-bit values, but not common on Alpha AXP and MIPS64 because those architectures naturally *sign*-extend 32-bit values to 64-bit values.

I find it amusing that the PowerPC, patron saint of ridiculous instructions, has an instruction whose name almost literally proclaims its ridiculousness: rldicl. (It stands for “rotate left doubleword by immediate and clear left”.)

For 32-bit processors with compiler support for multiword arithmetic, you end up with something like this:

```
// x86-32
    mov     eax, a          ; eax = a
    xor     ecx, ecx        ; Zero-extend to 64 bits
    add     eax, b          ; Accumulate low 32 bits in eax, set carry on overflow
    adc     ecx, ecx        ; Accumulate high 32 bits in ecx
                            ; ecx:eax = 64-bit result
    shrd    eax, ecx, 1     ; Multiword shift right
                            ; Answer in eax

// ARM 32-bit: Assume r0 = a, r1 = b
    mov     r2, #0          ; r2 = 0
    adds    r0, r1, r2      ; Accumulate low 32 bits in r0, set carry on overflow
    adc     r1, r2, #0      ; Accumulate high 32 bits in r1
                            ; r1:r0 = 64-bit result
    lsrs    r1, r1, #1      ; Shift high 32 bits right one position
                            ; Bottom bit goes into carry
    rrx     r0, r0          ; Rotate bottom 32 bits right one position
                            ; Carry bit goes into top bit
                            ; Answer in r0

// SH-3: Assume r4 = a, r5 = b
    ; (MSVC 13.10.3343 code generation here isn't that great)
    clrt                    ; Clear T flag
    mov     #0, r3          ; r3 = 0, zero-extended high 32 bits of a
    addc    r5, r4          ; r4 = r4 + r5 + T, overflow goes into T bit
    mov     #0, r2          ; r2 = 0, zero-extended high 32 bits of b
    addc    r3, r2          ; r2 = r2 + r3 + T, calculate high 32 bits
                            ; r3:r2 = 64-bit result
    mov     #31, r3         ; Prepare for left shift
    shld    r3, r2          ; r2 = r2 << r3
    shlr    r4              ; r4 = r4 >> 1
    mov     r2, r0          ; r0 = r2
    or      r4, r0          ; r0 = r0 | r4
                            ; Answer in r0

// MIPS: Assume a0 = a, a1 = b
    addu    v0, a0, a1      ; v0 = a0 + a1
    sltu    a0, v0, a0      ; a0 = 1 if overflow occurred
    sll     a0, 31          ; Move to bit 31
    srl     v0, v0, #1      ; Shift low 32 bits right one position
    or      v0, v0, a0      ; Combine the two parts
                            ; Answer in v0

// PowerPC: Assume r3 = a, r4 = b
    ; (gcc 4.8.5 -O3 code generation here isn't that great)
    mr      r9, r3          ; r9 = r3 (low 32 bits of 64-bit a)
    mr      r11, r4         ; r11 = r4 (low 32 bits of 64-bit b)
    li      r8, #0          ; r8 = 0 (high 32 bits of 64-bit a)
    li      r10, #0         ; r10 = 0 (high 32 bits of 64-bit b)
    addc    r11, r11, r9    ; r11 = r11 + r9, set carry on overflow
    adde    r10, r10, r8    ; r10 = r10 + r8, high 32 bits of 64-bit result
    rlwinm  r3, r10, 31, 1, 31 ; r3 = r10 >> 1
```

```
    rlwinm  r9, r11, 31, 0, 0 ; r9 = r1 << 31
    or      r3, r3, r9        ; Combine the two parts
                              ; Answer in r3

// RISC-V: Assume a0 = a, a1 = b
    add     a1, a0, a1        ; a1 = a0 + a1
    sltu    a0, a1, a0        ; a0 = 1 if overflow occurred
    slli    a0, a0, 31        ; Shift to bit 31
    slri    a1, a1, 1         ; a1 = a1 >> 1
    or      a0, a0, a1        ; Combine the two parts
                              ; Answer in a0
```

Or if you have access to SIMD registers that are larger than the native register size, you can do the math there. (Though crossing the boundary from general-purpose register to SIMD register and back may end up too costly.)

```
// x86-32
unsigned average(unsigned a, unsigned b)
{
    auto a128 = _mm_cvtsi32_si128(a);
    auto b128 = _mm_cvtsi32_si128(b);
    auto sum = _mm_add_epi64(a128, b128);
    auto avg = _mm_srli_epi64(sum, 1);
    return _mm_cvtsi128_si32(avg);
}

    movd    xmm0, a           ; Load a into bottom 32 bits of 128-bit register
    movd    xmm1, b           ; Load b into bottom 32 bits of 128-bit register
    paddq   xmm1, xmm0        ; Add as 64-bit integers
    psrlq   xmm1, 1           ; Shift 64-bit integer right one position
    movd    eax, xmm1         ; Extract bottom 32 bits of result

// 32-bit ARM (A32) has an "average" instruction built in
unsigned average(unsigned a, unsigned b)
{
    auto a64 = vdup_n_u32(a);
    auto b64 = vdup_n_u32(b);
    auto avg = vhadd_u32(a64, b64); // hadd = half of add (average)
    return vget_lane_u32(avg);
}

    vdup.32 d16, r0           ; Broadcast r0 into both halves of d16
    vdup.32 d17, r1           ; Broadcast r1 into both halves of d17
    vhadd.u32 d16, d16, d17 ; d16 = average of d16 and d17
    vmov.32 r0, d16[0]        ; Extract result
```

But you can still do better, if only you had access to better intrinsics.

In processors that support add-with-carry, you can view the sum of register-sized integers as a $(N + 1)$-bit result, where the bonus bit N is the carry bit. If the processor also supports rotate-right-through-carry, you can shift $(N + 1)$-bit result right one place, recovering the

correct average without losing the bit that overflows.

```
// x86-32
    mov     eax, a
    add     eax, b          ; Add, overflow goes into carry bit
    rcr     eax, 1          ; Rotate right one place through carry

// x86-64
    mov     rax, a
    add     rax, b          ; Add, overflow goes into carry bit
    rcr     rax, 1          ; Rotate right one place through carry

// 32-bit ARM (A32)
    mov     r0, a
    adds    r0, b           ; Add, overflow goes into carry bit
    rrx     r0              ; Rotate right one place through carry

// SH-3
    clrt                    ; Clear T flag
    mov     a, r0
    addc    b, r0           ; r0 = r0 + b + T, overflow goes into T bit
    rotcr   r0              ; Rotate right one place through carry
```

While there is an intrinsic for the operation of “add two values and report the result as well as carry”, we don’t have one for “rotate right through carry”, so we can get only halfway there:

```
unsigned average(unsigned a, unsigned b)
{
#if defined(_MSC_VER)
    unsigned sum;
    auto carry = _addcarry_u32(0, a, b, &sum);
    return _rotr1_carry(sum, carry); // missing intrinsic!
#elif defined(__clang__)
    unsigned carry;
    auto sum = _builtin_adc(a, b, 0, &carry);
    return _builtin_rotateright1throughcarry(sum, carry); // missing intrinsic!
#elif defined(__GNUC__)
    unsigned sum;
    auto carry = __builtin_add_overflow(a, b, &sum);
    return _builtin_rotateright1throughcarry(sum, carry); // missing intrinsic!
#else
#error Unsupported compiler.
#endif
}
```

We’ll have to fake it, alas. Here’s one way:

```
unsigned average(unsigned a, unsigned b)
{
#if defined(_MSC_VER)
    unsigned sum;
    auto carry = _addcarry_u32(0, a, b, &sum);
    return (sum / 2) | (carry << 31);
#elif defined(__clang__)
    unsigned carry;
    auto sum = _builtin_addc(a, b, 0, &carry);
    return (sum / 2) | (carry << 31);
#elif defined(__GNUC__)
    unsigned sum;
    auto carry = __builtin_add_overflow(a, b, &sum);
    return (sum / 2) | (carry << 31);
#else
#error Unsupported compiler.
#endif
}

// _MSC_VER
    mov     ecx, a
    add     ecx, b          ; Add, overflow goes into carry bit
    setc    al              ; al = 1 if carry set
    shr     ecx, 1          ; Shift sum right one position
    movzx   eax, al         ; eax = 1 if carry set
    shl     eax, 31         ; Move to bit 31
    or      eax, ecx        ; Combine
                            ; Result in eax

// __clang__
    mov     ecx, a
    add     ecx, b          ; Add, overflow goes into carry bit
    setc    al              ; al = 1 if carry set
    shld    eax, ecx, 31    ; Shift left 64-bit value
                            ; Result in eax

// __clang__ with ARM-Thumb2
    adds    r0, r0, r1      ; Calculate sum with flags
    blo     nope            ; Jump if carry clear
    movs    r1, #1          ; Carry is 1
    lsls    r1, r1, #31     ; Move carry to bit 31
    lsrs    r0, r0, #1      ; Shift sum right one position
    adcs    r0, r0, r1      ; Combine
    b       done
nope:
    movs    r1, #0          ; Carry is 0
    lsrs    r0, r0, #1      ; Shift sum right one position
    adds    r0, r0, r1      ; Combine
done:


// __GNUC__
```

```
    mov     eax, a
    xor     edx, edx        ; Preset edx = 0 for later setc
    add     eax, b          ; Add, overflow goes into carry bit
    setc    dl              ; dl = 1 if carry set
    shr     eax, 1          ; Shift sum right one position
    shl     edx, 31         ; Move carry to bit 31
    or      eax, edx        ; Combine
```

I considered trying a sneaky trick: Use the rotation intrinsic. (gcc doesn't have a rotation intrinsic, so I couldn't try it there.)

```
unsigned average(unsigned a, unsigned b)
{
#if defined(_MSC_VER)
    unsigned sum;
    auto carry = _addcarry_u32(0, a, b, &sum);
    sum = (sum & ~1) | carry;
    return _rotr(sum, 1);
#elif defined(__clang__)
    unsigned carry;
    sum = (sum & ~1) | carry;
    auto sum = __builtin_addc(a, b, 0, &carry);
    return __builtin_rotateright32(sum, 1);
#else
#error Unsupported compiler.
#endif
}

// _MSC_VER
    mov     ecx, a
    add     ecx, b          ; Add, overflow goes into carry bit
    setc    al              ; al = 1 if carry set
    and     ecx, -2         ; Clear bottom bit
    movzx   ecx, al         ; Zero-extend byte to 32-bit value
    or      eax, ecx        ; Combine
    ror     ear, 1          ; Rotate right one position
                            ; Result in eax

// __clang__
    mov     ecx, a
    add     ecx, b          ; Add, overflow goes into carry bit
    setc    al              ; al = 1 if carry set
    shld    eax, ecx, 31    ; Shift left 64-bit value

// __clang__ with ARM-Thumb2
    movs    r2, #0          ; Prepare to receive carry
    adds    r0, r0, r1      ; Calculate sum with flags
    adcs    r2, r2          ; r2 holds carry
    lsrs    r0, r0, #1      ; Shift sum right one position
    lsls    r1, r2, #31     ; Move carry to bit 31
    adds    r0, r1, r0      ; Combine
```

Mixed results. For `_MSC_VER`, the code generation got worse. For `__clang__` for ARM-Thumb2, the code generation got better. And for `__clang__` for x86, the compiler realized that it was the same as before, so it just used the previous codegen!

Bonus chatter: And while I'm here, here are sequences for processors that don't have rotate-right-through-carry.

```
// AArch64 (A64)
    mov     x0, a
    adds    x0, x1, b       ; Add, overflow goes into carry bit
    addc    x1, xzr, xzr    ; Copy carry to x1
    extr    x0, x1, x0, 1   ; Extract bits 64:1 from x1:x0
                            ; Answer in x0

// Alpha AXP: Assume a0 = a, a1 = b, both 64-bit values
    addq    a0, a1, v0      ; 64-bit addition: v0 = a0 + a1
    cmpult  a0, v0, a0      ; a0 = 1 if overflow occurred
    srl     v0, #1, v0      ; 64-bit shift:    v0 = v0 >> 1
    sll     a0, #63, a0     ; 64-bit shift:    a0 = a0 << 63
    or      a0, v0, v0      ; v0 = v0 | a0
                            ; Answer in v0

// Itanium Ia64: Assume r32 = a, r33 = b, both 64-bit values
    add     r8 = r32, r33 ;;    // 64-bit addition: r8 = r32 + r33
    cmp.ltu p6, p7 = r8, r33 ;; // p6 = true if overflow occurred
(p6) addl   r9 = 1, r0          // r9 = 1 if overflow occurred
(p7) addl   r9 = 0, r0 ;;       // r9 = 0 if overflow did not occur
    shrp    r8 = r9, r8, 1      // r8 = extract bits 64:1 from r9:r8
                                // Answer in r8

// MIPS: Same as multiprecision version

// PowerPC: Assume r3 = a, r4 = b
    addc    r3, r3, r4          ; Accumulate low 32 bits in r3, set carry on overflow
    adde    r5, r4, r4          ; Shift carry into bottom bit of r5 (other bits
garbage)
    rlwinm  r3, r3, 31, 1, 31   ; Shift r3 right by one position
    rlwinm  r5, r5, 31, 0, 0    ; Shift bottom bit of r5 to bit 31
    or      r3, r5, r5          ; Combine the two parts

// RISC-V: Same as multiprecision version
```

Bonus chatter: C++20 adds a `std::midpoint` function that calculates the average of two values (rounding toward `a`).

Bonus viewing: std::midpoint? How hard could it be?

Update: I was able to trim an instruction off the PowerPC version by realizing that only the bottom bit of `r5` participates in the `rlwinm`, so the other bits can be uninitialized garbage. For the uninitialized garbage, I used `r4`, which I know can be consumed without a stall

because the `addc` already consumed it.

Here's the original:

```
// PowerPC: Assume r3 = a, r4 = b
    li      r5, #0              ; r5 = 0 (accumulates high 32 bits)
    addc    r3, r3, r4          ; Accumulate low 32 bits in r3, set carry on overflow
    addze   r5, r5              ; Accumulate high bits in r5
    rlwinm  r3, r3, 31, 1, 31   ; Shift r3 right by one position
    rlwinm  r5, r5, 31, 0, 0    ; Shift bottom bit of r5 to bit 31
    or      r3, r5, r5          ; Combine the two parts
```

Update 2: Peter Cordes pointed out that an instruction can also be trimmed from the AArch64 version by using the `uxtw` extended register operation to combine a `uxtw` with an `add`. Here's the original:

```
// AArch64 (ARM 64-bit): Assume w0 = a, w1 = b, upper 32 bits unknown
    uxtw    x0, w0          ; x0 = w0 zero-extended to 64-bit value
    uxtw    x1, w1          ; x1 = w1 zero-extended to 64-bit value
    add     x0, x1          ; 64-bit addition: x0 = x0 + x1
    ubfx    x0, x0, 1, 32   ; Extract bits 1 through 32 from result
                            ; (shift + zero-extend in one instruction)
                            ; Answer in x0
```

Raymond Chen

Follow