Negation: Complement & Increment

- **Claim: Following Holds for 2's Complement**

  `~x + 1 == -x`

- **Complement**
  - Observation: `~x + x == 1111…111 == -1`

	x	1 0 0 1 1 1 0 1
+	~x	0 1 1 0 0 0 1 0
	-1	1 1 1 1 1 1 1 1

Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Expanding the Bit Representation of a Number:

One common operation is to convert between integers having different word sizes while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible.

- ✓ To convert an unsigned number to a larger data type, we can simply add leading zeros to the representation; this operation is known as zero extension
- ✓ For converting a two's-complement number to a larger data type, the rule is to perform a sign extension, adding copies of the most significant bit to the representation

Apply this eqn below to solve the next problem

PRINCIPLE: Definition of two's-complement encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

Show that each of the following bit vectors is a two's-complement representation of −4

A. [1100]
B. [11100]
C. [111100]
Observe that the second and third bit vectors can be derived from the first by sign extension.

Unsigned Addition

Let us define the operation $+_w^u$ for arguments x and y, where $0 \le x, y < 2^w$, as the result of truncating the integer sum $x + y$ to be w bits long and then viewing the result as an unsigned number. This can be characterized as a form of modular arithmetic, computing the sum modulo 2^w by simply discarding any bits with weight greater than 2^{w-1} in the bit-level representation of $x + y$. For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations [1001] and [1100], respectively. Their sum is 21, having a 5-bit representation [10101]. But if we discard the high-order bit, we get [0101], that is, decimal value 5. This matches the value 21 mod 16 = 5.

PRINCIPLE: Unsigned addition

For x and y such that $0 \le x, y < 2^w$:

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \quad \text{Normal} \\ x + y - 2^w, & 2^w \le x + y < 2^{w+1} \quad \text{Overflow} \end{cases}$$

In general, we can see that if $x + y < 2^w$, the leading bit in the $(w + 1)$-bit representation of the sum will equal 0, and hence discarding it will not change the numeric value. On the other hand, if $2^w \leq x + y < 2^{w+1}$, the leading bit in the $(w + 1)$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. ■

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type

When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether or not overflow has occurred.

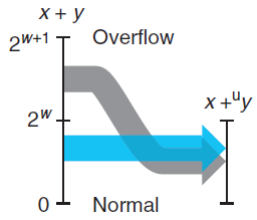
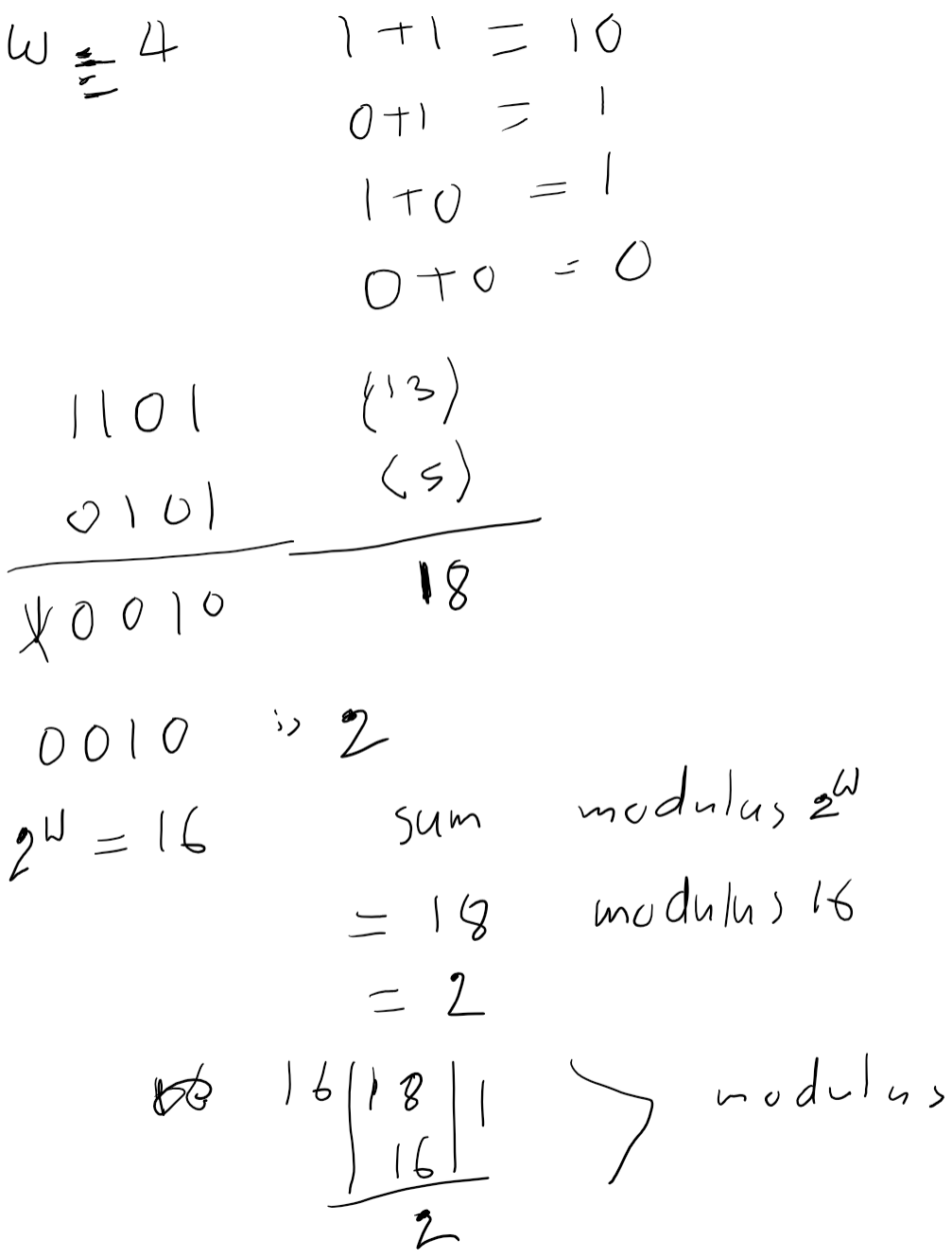


Fig: Relation between integer addition and unsigned addition. When x + y is greater than $2^w - 1$, the sum overflows.

Example:



For this example, the word size is 4 (you have 4 bits to represent your number). When you add 13 with 5, the result is 18, which is 5-bits in binary. The original result is 10010. But, you can't fit it because we have only 4-bits allocated. Therefore, discard the MSB bit, now you get 0010 which is 2. If you do modular arithmetic => sum modulus 2^w, then 18 modulus 16 is 2 too.

Learn more about modular arithmetic https://en.wikipedia.org/wiki/Modular_arithmetic

Signed Addition

With two's-complement addition, we must decide what to do when the result is either too large (positive) or too small (negative) to represent. Given integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$, their sum is in the range $-2^w \le x + y \le 2^w - 2$, potentially requiring $w + 1$ bits to represent exactly. As before, we avoid ever-expanding data sizes by truncating the representation to w bits. The result is not as familiar mathematically as modular addition, however. Let us define $x +^t_w y$ to be the result of truncating the integer sum $x + y$ to be w bits long and then viewing the result as a two's-complement number.

PRINCIPLE: Two's-complement addition

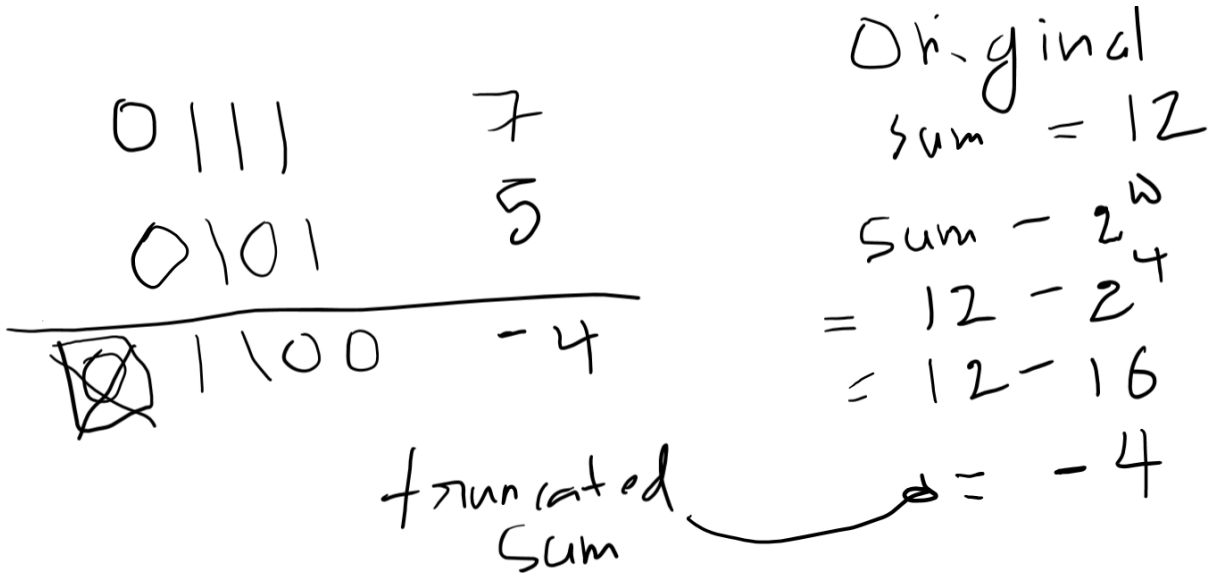
For integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$:

$$x +^t_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative overflow} \end{cases}$$

Positive Overflow:

When the sum $x + y$ exceeds *TMaxw*, we say that *positive overflow* has occurred. In this case, the effect of truncation is to subtract 2^w from the sum.

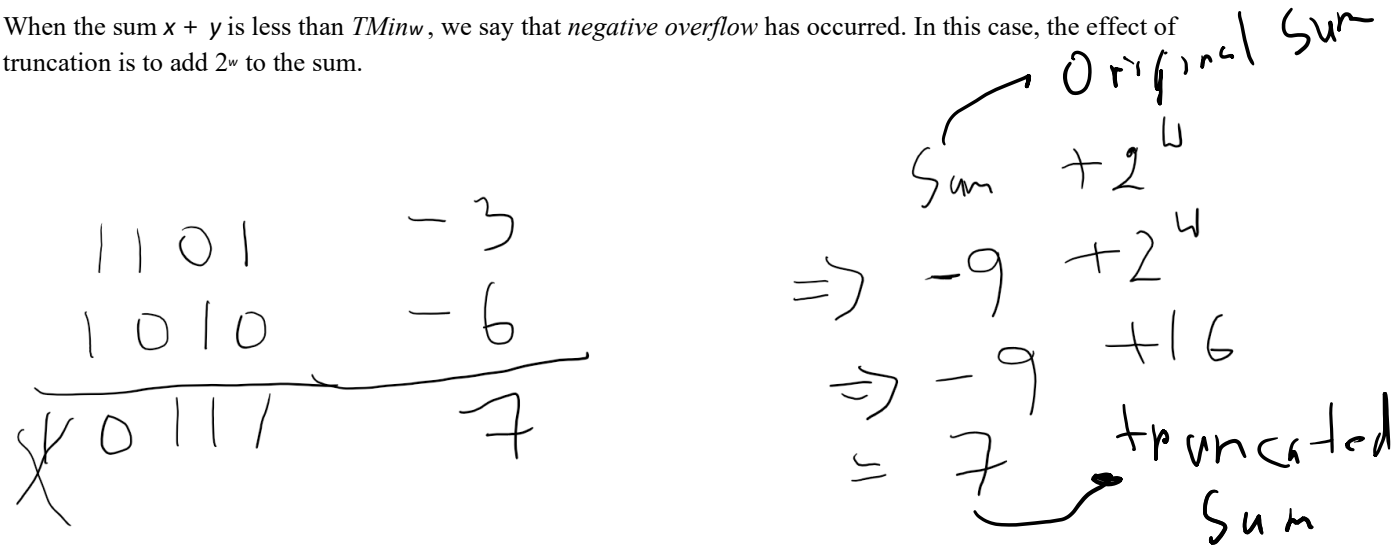
So, you can see that you can add 7 with 5, the original sum will be 12. But you cannot represent 12 with 4-bits. Therefore you do truncation, discard one bit, then you get 1100, 1100 means -4. Here, -4 is your truncated sum and this is a case of positive overflow. Here, you can explain the truncated sum by applying this formula below:
Original sum – 2^w



When you add two positive numbers, but get a negative result, that's a positive overflow.

Negative Overflow:

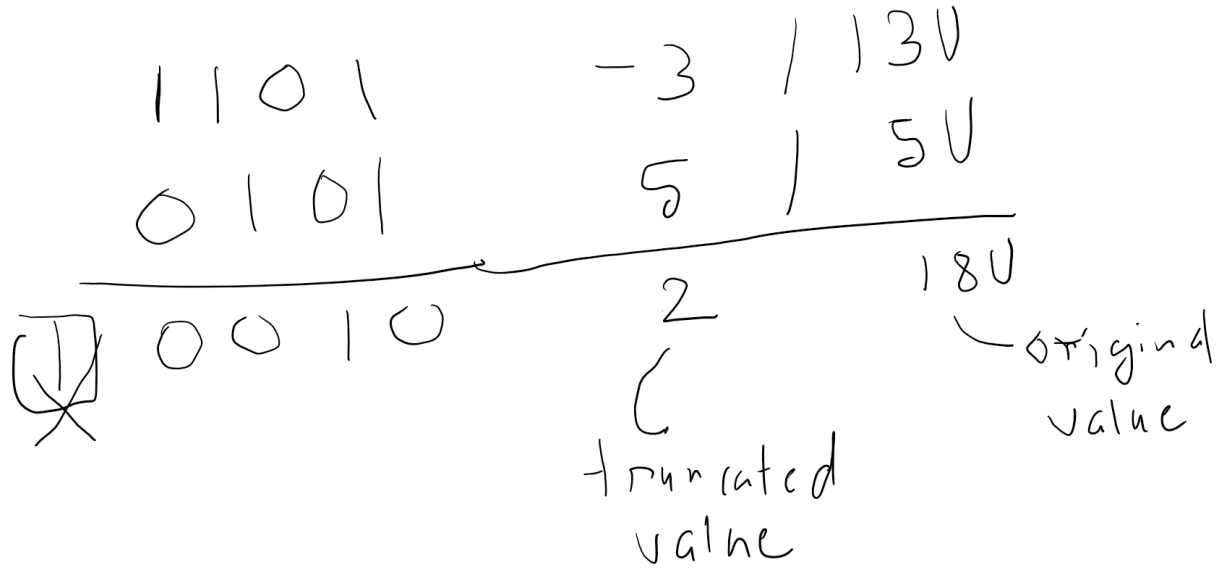
When the sum $x + y$ is less than $TMin_w$, we say that *negative overflow* has occurred. In this case, the effect of truncation is to add 2^w to the sum.



Here, the original sum is -9. But, you cannot represent -9 with 4-bits. Therefore, we truncate one bit and we get 0111 as our result. But, 0111 means 7. If you add original sum with 2^w, then you get the truncated sum.

When you add two negative numbers, but get a positive result, that's a negative overflow

*****Show that bit-level representation is same for signed and unsigned addition, even though the values are different.**



For this example, you can see that the truncated sum is 2, this result is correct when you consider that for signed. If you try to interpret this result as unsigned, then the original sum is 10010, which is 18. And 18 modulo 2^w is 2. Here, 2 is the truncated sum for unsigned addition. But, again the bit-level representation are same, but the value/interpretation are different for signed and unsigned.

PRINCIPLE: Detecting overflow of unsigned addition

For x and y in the range $0 \le x, y \le UMax_w$, let $s \doteq x +^u_w y$. Then the computation of s overflowed if and only if $s < x$ (or equivalently, $s < y$). ■

DERIVATION: Detecting overflow of unsigned addition

Observe that $x + y \geq x$, and hence if s did not overflow, we will surely have $s \geq x$. On the other hand, if s did overflow, we have $s = x + y - 2^w$. Given that $y < 2^w$, we have $y - 2^w < 0$, and hence $s = x + (y - 2^w) < x$. ■

PRINCIPLE: Detecting overflow in two's-complement addition

For x and y in the range $TMin_w \leq x, y \leq TMax_w$, let $s \doteq x +^{\mathrm{t}}_w y$. Then the computation of s has had positive overflow if and only if $x > 0$ and $y > 0$ but $s \leq 0$. The computation has had negative overflow if and only if $x < 0$ and $y < 0$ but $s \geq 0$. ■

DERIVATION: Detecting overflow of two's-complement addition

Let us first do the analysis for positive overflow. If both $x > 0$ and $y > 0$ but $s \leq 0$, then clearly positive overflow has occurred. Conversely, positive overflow requires (1) that $x > 0$ and $y > 0$ (otherwise, $x + y < TMax_w$) and (2) that $s \leq 0$ (from Equation 2.13). A similar set of arguments holds for negative overflow. ■

The w-bit two's-complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed addition

Unsigned Multiplication

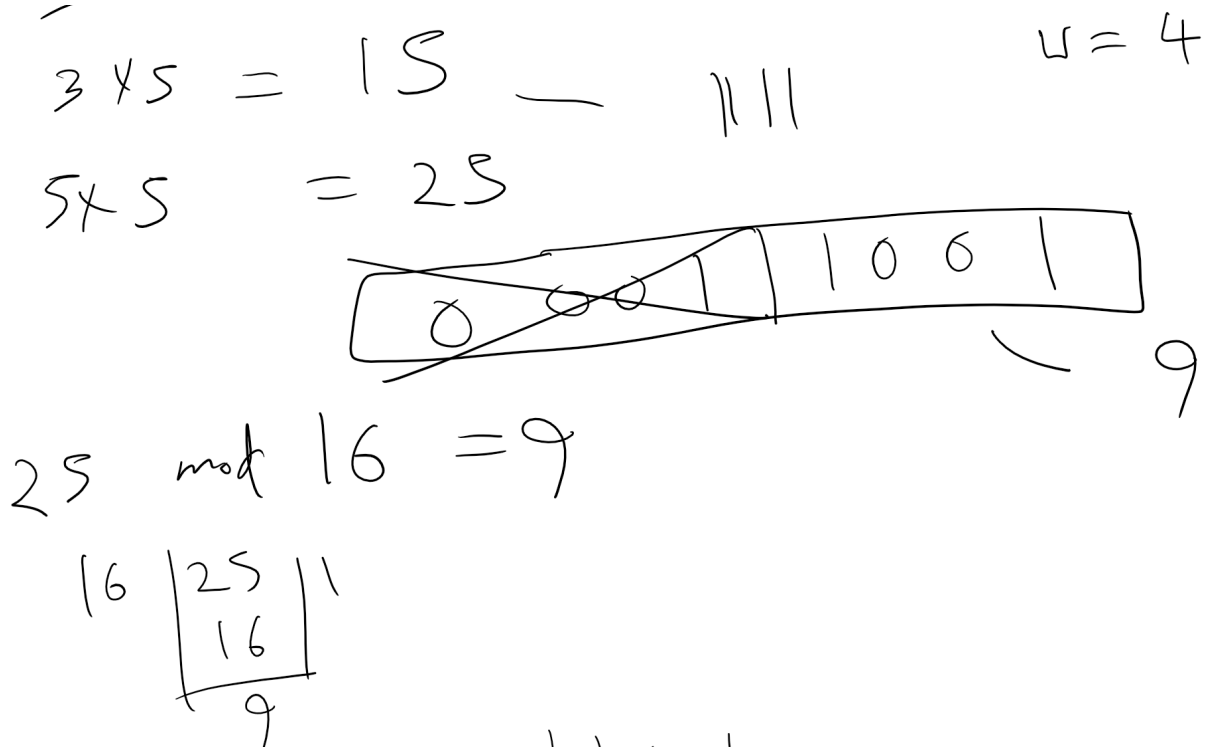
Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. This could require as many as $2w$ bits to represent. Instead, unsigned multiplication in C is defined to yield the w-bit value given by the low-order w bits of the $2w$-bit integer product. Let us denote this value as $x *^{\mathrm{u}}_w y$.

Truncating an unsigned number to w bits is equivalent to computing its value modulo 2^w, giving the following:

PRINCIPLE: Unsigned multiplication

For x and y such that $0 \leq x, y \leq UMax_w$:

$$x *^{\mathrm{u}}_w y = (x \cdot y) \bmod 2^w$$



Here, you are multiplying 5 with 5, and the original result is 25. But, you can cannot represent 25 with 4-bits. We are only considering the lower 4-bits, and that's 1001. The truncated result is 9. If you do original result mod 2^w, then you will get 9.

Signed Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's-complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$. This could require as many as $2w$ bits to represent in two's-complement form. Instead, signed multiplication in C generally is performed by truncating the $2w$-bit product to w bits. We denote this value as $x *_w^t y$. Truncating a two's-complement number to w bits is equivalent to first computing its value modulo 2^w and then converting from unsigned to two's complement, giving the following:

PRINCIPLE: Two's-complement multiplication

For x and y such that $TMin_w \leq x, y \leq TMax_w$:

$$x *_w^t y = U2T_w((x \cdot y) \bmod 2^w) \tag{2.17}$$

We claim that the bit-level representation of the product operation is identical for both unsigned and two's-complement multiplication

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's complement	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's complement	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's complement	3	[011]	3	[011]	9	[001001]	1	[001]

Three-bit unsigned and two's-complement multiplication examples.
Although the bit-level representations of the full products may differ, those of the truncated products are identical.

Multiplying by Constants

Historically, the integer multiply instruction on many machines was fairly slow, requiring 10 or more clock cycles, whereas other integer operations—such as addition, subtraction, bit-level operations, and shifting—required only 1 clock cycle. Even on the Intel Core i7 Haswell we use as our reference machine, integer multiply requires 3 clock cycles. As a consequence, one important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations.

PRINCIPLE: Multiplication by a power of 2

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, the $w+k$-bit unsigned representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k zeros have been added to the right. ■

So, for example, 11 can be represented for $w = 4$ as [1011]. Shifting this left by $k = 2$ yields the 6-bit vector [101100], which encodes the unsigned number $11 \cdot 4 = 44$.

shifting a value left is equivalent to performing unsigned multiplication by a power of 2

PRINCIPLE: Unsigned multiplication by a power of 2

For C variables `x` and `k` with unsigned values x and k, such that $0 \leq k < w$, the C expression `x << k` yields the value $x *_w^{\mathrm{u}} 2^k$. ■

Since the bit-level operation of fixed-size two's-complement arithmetic is equivalent to that for unsigned arithmetic, we can make a similar statement about the relationship between left shifts and multiplication by a power of 2 for two's-complement arithmetic:

PRINCIPLE: Two's-complement multiplication by a power of 2

For C variables `x` and `k` with two's-complement value x and unsigned value k, such that $0 \leq k < w$, the C expression `x << k` yields the value $x *_w^{\mathrm{t}} 2^k$. ■

Note that multiplying by a power of 2 can cause overflow with either unsigned or two's-complement arithmetic. Our result shows that even then we will get the same effect by shifting. Returning to our earlier example, we shifted the 4-bit pattern [1011] (numeric value 11) left by two positions to get [101100] (numeric value 44). Truncating this to 4 bits gives [1100] (numeric value $12 = 44 \bmod 16$).

Given that integer multiplication is more costly than shifting and adding, many C compilers try to remove many cases where an integer is being multiplied by a constant with combinations of shifting, adding, and subtracting. For example, suppose a program contains the expression `x*14`. Recognizing that $14 = 2^3 + 2^2 + 2^1$, the compiler can rewrite the multiplication as `(x<<3) + (x<<2) + (x<<1)`, replacing one multiplication with three shifts and two additions. The two computations will yield the same result, regardless of whether `x` is unsigned or two's complement, and even if the multiplication would cause an overflow. Even better, the compiler can also use the property $14 = 2^4 - 2^1$ to rewrite the multiplication as `(x<<4) - (x<<1)`, requiring only two shifts and a subtraction.

Dividing by Powers of 2

Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles. Dividing by a power of 2 can also be performed using shift operations, but we use a right shift rather than a left shift. The two different right shifts—logical and arithmetic—serve this purpose for unsigned and two's-complement numbers, respectively.

The case for using shifts with unsigned arithmetic is straightforward, in part because right shifting is guaranteed to be performed logically for unsigned values.

PRINCIPLE: Unsigned division by a power of 2

For C variables `x` and `k` with unsigned values x and k, such that $0 \leq k < w$, the C expression `x >> k` yields the value $\lfloor x/2^k \rfloor$. ■

k	>> k (binary)	Decimal	$12{,}340/2^k$
0	0011000000110100	12,340	12,340.0
1	*0*001100000011010	6,170	6,170.0
4	*0000*001100000011	771	771.25
8	*00000000*00110000	48	48.203125

Dividing unsigned numbers by powers of 2. The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

examples show the effects of performing logical right shifts on a 16-bit representation of 12,340 to perform division by 1, 2, 16, and 256. The zeros shifted in from the left are shown in italics.We also show the result we would obtain if we did these divisions with real arithmetic. These examples show that the result of shifting consistently rounds toward zero, as is the convention for integer division.

Two's complement division : arithmetic right shift

k	>> k (binary)	Decimal	$-12{,}340/2^k$
0	1100111111001100	−12,340	−12,340.0
1	1110011111100110	−6,170	−6,170.0
4	1111110011111100	−772	−771.25
8	1111111111001111	−49	−48.203125

****** Pick up different integers, and perform addition/multiplication/division etc. , both for signed and unsigned.**