Computer arithmetic (Contd.)

Recall: Signed representation (Two's complement)

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

T_{min} T_{max}

limits of representation

Unsigned representation:

$$B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i \cdot 2^i .$$

U_{max}

$B2T_w(\vec{x})$ and $B2U_w(\vec{x})$ are bijections.

$\Rightarrow$ $T2B_w$ is defined to be $B2T_w^{-1}$

$U2B_w$ $B2U_w^{-1}$ $\Big\}$.

Thus we can get $U2T_w(y) = B2T_w(U2B_w(y))$.

Qn: What happens if we convert a negative value to unsigned int.

Ex: (unsigned short) $-12345 = 53191$.

Qn: What happens if we convert _-bit unsigned int to signed representation.

32 bit $u = 4294967295$

$\downarrow$ (int)

-1

Obs: Numbers between 0 to 2^{w-1} can be converted without any issues.

Signed representation

$$-x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \Big\}$$

From previous example:

$$65536 + (-12345) = 53191$$

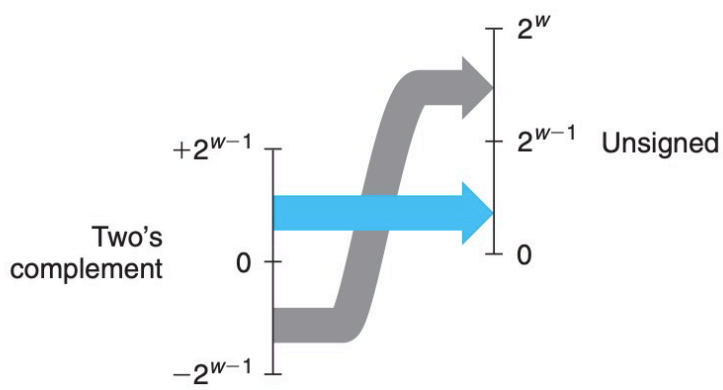
$$-12345 = -65536 + 53191$$

Fix Binary represention $\vec{x}$

Difference between the unsigned value and signed value is $x_{w-1} \cdot 2^w$.

$$B2U(\vec{x}) = x_{w-1} \cdot 2^w + B2T(\vec{x}).$$

Figure 2.16
Conversion from two's complement to unsigned. Function $T2U$ converts negative numbers to large positive numbers.



$$T2U_w(x) = B2U(T2B_w(x)) = x_{w-1} \cdot 2^w + x$$

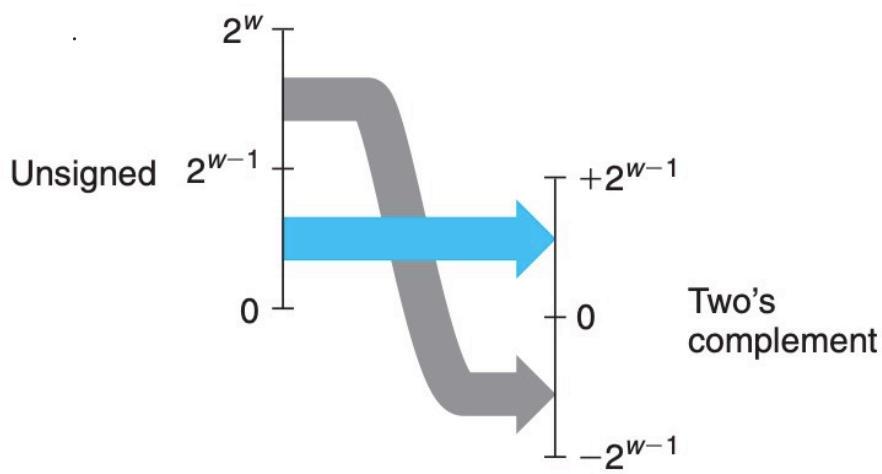
$$T2U_w(x) = \begin{cases} x + 2^w & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Similarly,

$$U2T_w(u) = B2T_w(U2B_w(u)) = -u_{w-1} \cdot 2^w + u.$$

$$U2T_w(u) = \begin{cases} u & u < 2^{w-1} \\ u - 2^w & u \geq 2^{w-1} \end{cases}$$

Figure 2.17
Conversion from unsigned to two's complement. Function *U2T* converts numbers greater than $2^{w-1} - 1$ to negative values.




```
int x = -1;
unsigned u = 2147483648; /* 2 to the 31st */

printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine, it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

Qn: So far we only saw conversion between same word sizes. What about different word sizes?

!! Conversion from large word size to small word size can lead to messy situations.

But we convert small word sizes to larger word sizes.

→ Unsigned:

→ Signed:

$$[x_{w-1}, x_{w-2}, \dots, x_0] \longrightarrow [x_{w'-1}, x_{w'-2}, \dots, x_{w-1}, \dots, x_0]$$

Claim: $x_{w'-1} = x_{w'-2} = \dots = x_w = x_{w-1}$.

In other words,

$$B2T_{w+k}\Big([\underbrace{x_{w-1}, x_{w-1}, \dots, x_{w-1}}_{k \text{ times}}, \dots, x_0]\Big) = B2T_w\big([x_{w-1}, \dots, x_0]\big).$$

Truncating numbers

$$[x_{w-1}, x_{w-2}, \dots, x_0] \longrightarrow [x_{k-1}, x_{k-2}, \dots, x_0]$$

$$B2U_w\big([x_{w-1}, x_{w-2}, \dots, x_0]\big) \bmod 2^k = B2U_k\big([x_{k-1}, \dots, x_0]\big).$$

Similarly,

$$B2T_w\big([x_{w-1}, x_{w-2}, \dots, x_0]\big) \bmod 2^k = B2U_k\big([x_{k-1}, \dots, x_0]\big).$$

$$B2T_k([x_{k-1}, \ldots, x_0]) = U2T_k(B2U_w[x_{w-1}, \ldots, x_0] \bmod 2^k)$$

Aside Security vulnerability in `getpeername`

In 2002, programmers involved in the FreeBSD open source operating systems project realized that their implementation of the `getpeername` library function had a security vulnerability. A simplified version of their code went something like this:

```
/*
 * Illustration of code vulnerability similar to that found in
 * FreeBSD's implementation of getpeername()
 */

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

Integer arithmetic

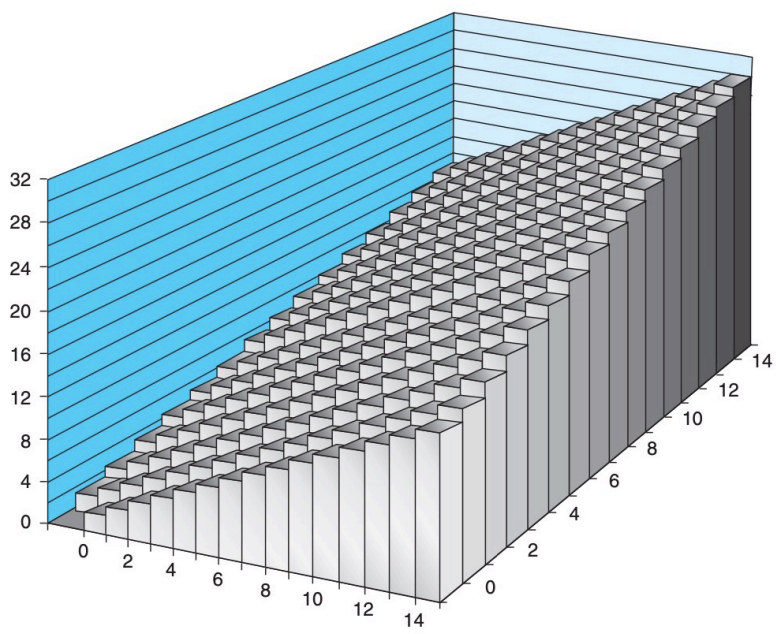
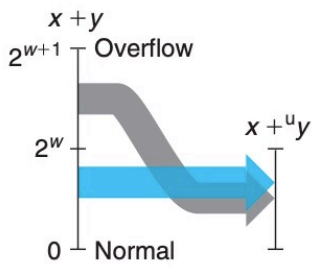


Figure 2.20 **Integer addition.** With a 4-bit word size, the sum could require 5 bits.

Figure 2.21
Relation between integer addition and unsigned addition. When $x + y$ is greater than $2^w - 1$, the sum overflows.



$$x +_w^u y = \begin{cases} x+y & x+y < 2^w \\ x+y-2^w & 2^w \le x+y < 2^{w+1}. \end{cases}$$

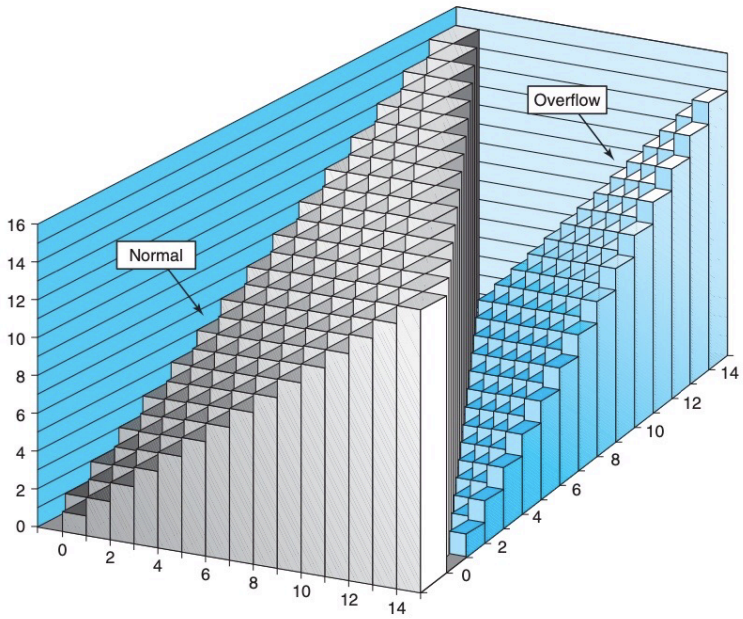


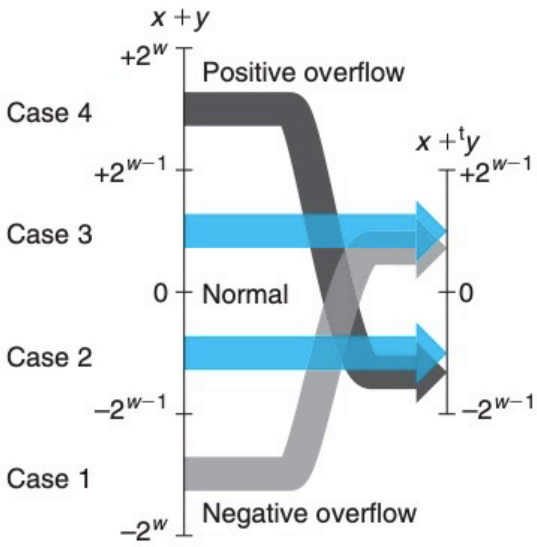
Figure 2.22 **Unsigned addition.** With a 4-bit word size, addition is performed modulo 16.

Qn: What about subtraction?

Two's complement representation

$$x +_w^t y = U2T_w\left(T2U_w(x) +_w^u T2U_w(y)\right)$$

Figure 2.23
Relation between integer and two's-complement addition. When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than $2^{w-1} + 1$, there is a positive overflow.



$$x +_w^t y = \begin{cases} x+y-2^w & \text{if } x+y \geq 2^{w-1} \\ x+y & \text{if } -2^{w-1} \leq x+y < 2^{w-} \\ x+y+2^w & \text{if } x+y < -2^{w-1} \end{cases}$$

$$-_w^t x = \begin{cases} -2^{w-1} & \text{if } x = -2^{w-1} \\ -x & \text{if } x > -2^{w-1}. \end{cases}$$

Integer multiplication

Unsigned $\quad x \times_w^u y = (x \cdot y) \bmod 2^w.$

Claim: Multiplication is identical

Given $\vec{x}$ and $\vec{y}$, let

$$x = B2T_w(\vec{x}) \quad , \quad y = B2T_w(\vec{y})$$

$$x' = B2U_w(\vec{x}) \quad , \quad y' = B2U_w(\vec{y}).$$

From prev discussion,

$$x' = x + x_{w-1} \cdot 2^w \quad \text{and} \quad y' = y + y_{w-1} \cdot 2^w.$$

$$(x' \cdot y') \bmod 2^w = \left[(x + x_{w-1} \cdot 2^w) \cdot (y + y_{w-1} \cdot 2^w) \right] \bmod 2^w$$

$$= (x \cdot y) \bmod 2^w.$$