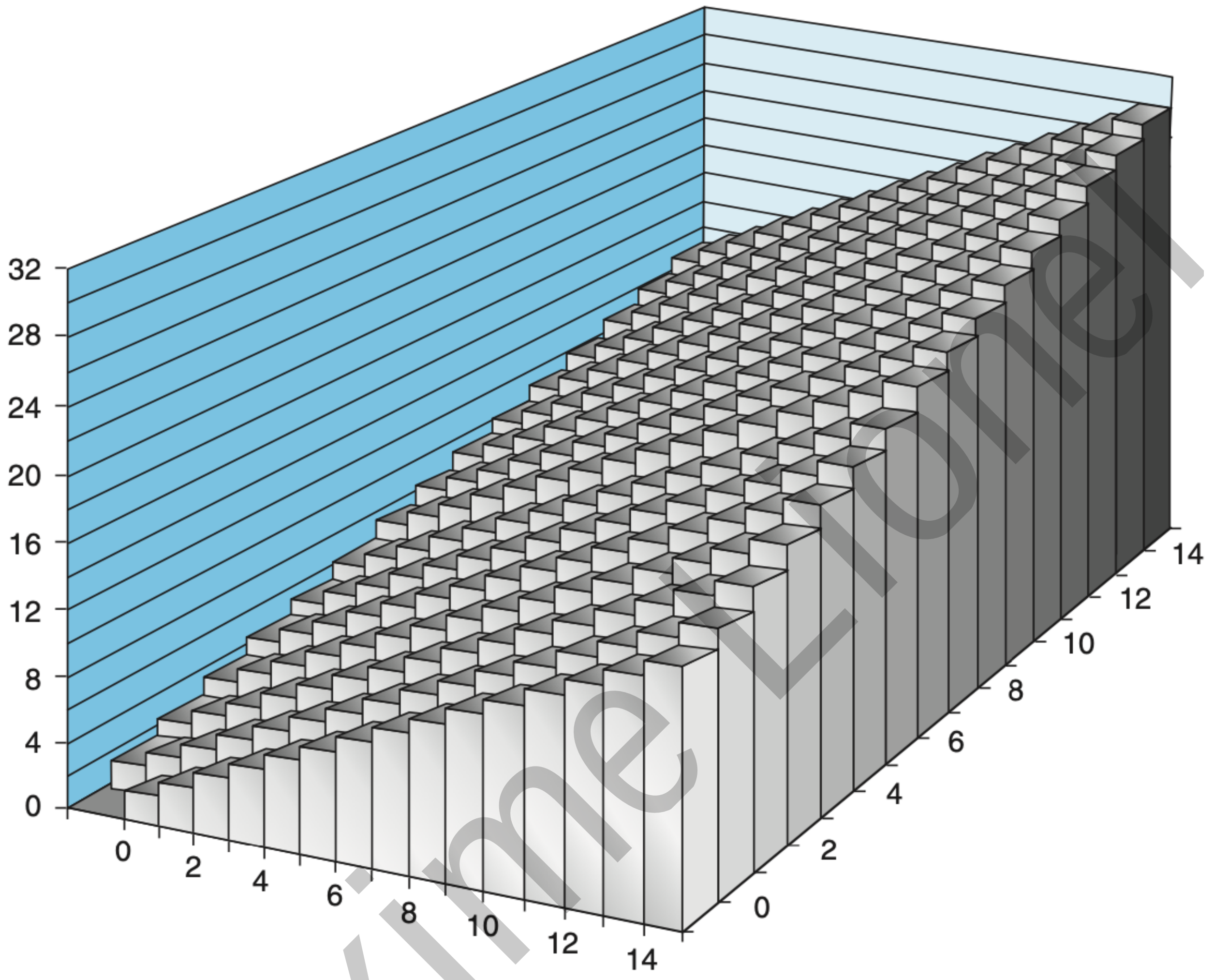
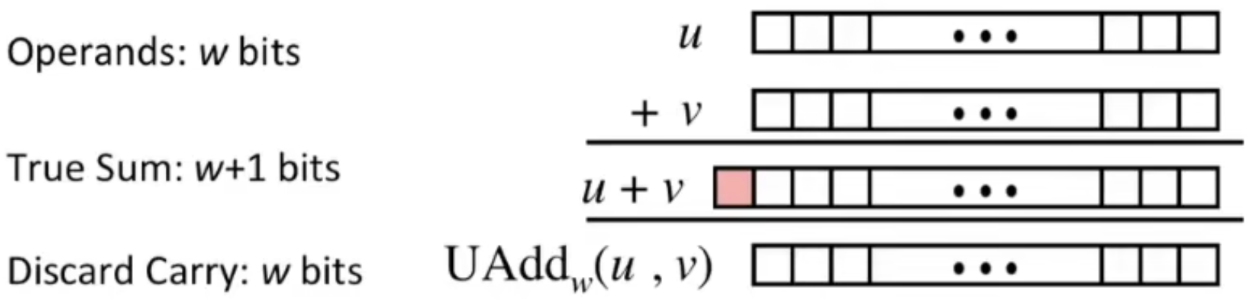
2_3

2.3 Integer Arithmetic

2.3.1 Unsigned Addition



- Example:
  - Suppose: w = 4 bits, $0 \le x, y \le 2^w - 1$
  - Then $0 \le x + y \le 2^{w+1} - 2$ and it requires w+1 bits
  - So when w = 4 bits, we get $0 \le x + y \le 30$ as the figure
  - To summarize, most programming languages support fixed-size arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.
- Unsigned addition:



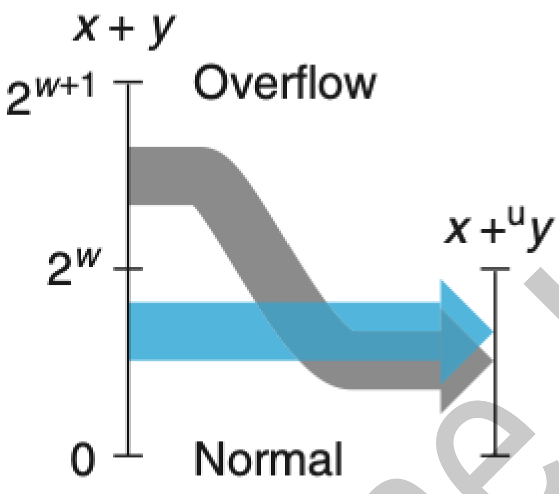
  - Define operation $+_w^u$ (u means unsigned, w means bit width), for $0 \le x, y < 2^w$, then $x +_w^u y$ to be w bits long and an unsigned number.

- Example:
  - Suppose x = 9 and y = 12, w = 4
  - Then x = [1001] and y = [1100]
  - Usually x+y = [10101]
  - Now $x +^u_4y = (x+y)%2^4 $ = [0101] = 5
- Fomula:
  - If $0 \leq x, y < 2^w$ and $x + y < 2^w$ - Normal Condition

    Then $x +^u_w y = x + y$
  - If $0 \leq x, y < 2^w$ and $2^w \leq x + y < 2^{w+1}$ - Overflow Condition

    Then $x +^u_w y = x + y - 2^w$
  - Combine 2 scenarios, we can write as below:

```
$x+^u_wy = (x+y)mod2^w$
```



- Back to example:
  - If x + y < 16, then normal (no overflow), so $x +^u_w y = x + y$
  - If $x + y \geq 16$, then overflow, so $x +^u_w y = x + y - 2^4$
- When executing C programs, overflows are not signaled as errors.
- How to detect overflow of unsigned addition?
  - Suppose $0 \leq x, y \leq UMax_w$
  - Then $s = x +^u_w y$
  - **If s < x or s < y, then overflow.**
  - Example:
    - w = 4, x = 9, y = 12
    - $s = x +^u_w y = 9 +^u_4 12 = 9 + 12 - 2^4 = 5$
    - We find s < x, so overflow!

Practice Problem 2.27

Write a function with the following prototype:

```
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y);
```

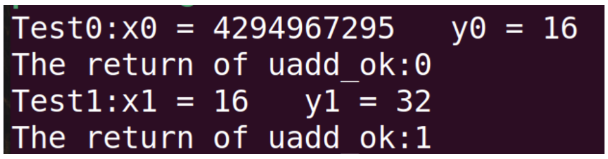
This function should return 1 if arguments x and y can be added without causing overflow.

```c
// P118.c
#include<stdio.h>
#include<limits.h>

int uadd_ok(unsigned x, unsigned y)
{
        unsigned s = x + y;
  if(s < x) return 0;
  return 1;
}

int main()
{
  unsigned x0 = UINT_MAX;
  unsigned y0 = 0x10;
  printf("Test0:");
  printf("x0 = %u\ty0 = %u\n",x0,y0);
  printf("The return of uadd_ok:%d\n",uadd_ok(x0,y0));

  unsigned x1 = 0x10;
  unsigned y1 = 0x20;
  printf("Test1:");
  printf("x1 = %u\ty1 = %u\n",x1,y1);
  printf("The return of uadd_ok:%d\n",uadd_ok(x1,y1));
  return 0;
}
```



```
Test0:x0 = 4294967295    y0 = 16
The return of uadd_ok:0
Test1:x1 = 16    y1 = 32
The return of uadd ok:1
```

- Modular addition (模数加法) forms a mathematical structure known as an **abelian group** (阿贝尔群):
  - commutative and associative (交换律和结合律)
  - Include an identity element 0 and each element has an additive inverse (加法逆元)
  - For each value x, there must be $-^{u}_{w}x$ which makes $-^{u}_{w}x +^{u}_{w} x = 0$
- Unsigned Negation (无符号取非):
  - Suppose $0 \leq x < 2^w$
  - Its w-bit unsigned negation $-^{u}_{w}x$ is:
    - if x==0, $-^{u}_{w}x = x$
    - If x > 0, $-^{u}_{w}x = 2^w - x$
  - Derivation:
    - if x ==0 , obviously its additive inverse is 0
    - if x > 0, to verify, we use:

      $-^{u}_{w}x +^{u}_{w} x$

      $= (2^w - x) +^{u}_{w} x$

      $= (2^w - x + x)\%2^w = 0$

      So, $-^{u}_{w}x$ is the additive inverse of x

Practice Problem 2.28

We can represent a bit pattern of length w = 4 with a single hex digit. For an unsigned interpretation of these digits, use Equation below to fill in the following table giving the values and the bit representations (in hex) of the unsigned additive inverses of the digits shown.

- Suppose $0 \leq x < 2^w$
- Its w-bit unsigned negation $-^{u}_{w}x$ is:
  - if x==0, $-^{u}_{w}x = x$

- If x > 0, $-^u_w x = 2^w - x$

x		$-^u_4 x$	
Hex	Decimal	Decimal	Hex
1			
4			
7			
A			
E			

Solution:

we know w = 4

x - Hex	x -Decimal	$-^u_4 x$ - Decimal	$-^u_4 x$ - Hex
1	1	15	F
4	4	12	C
7	7	9	9
A	10	6	6
E	14	2	2

1: $-^u_4 x = 2^4 - 1 = 15$

4: $-^u_4 x = 2^4 - 4 = 12$

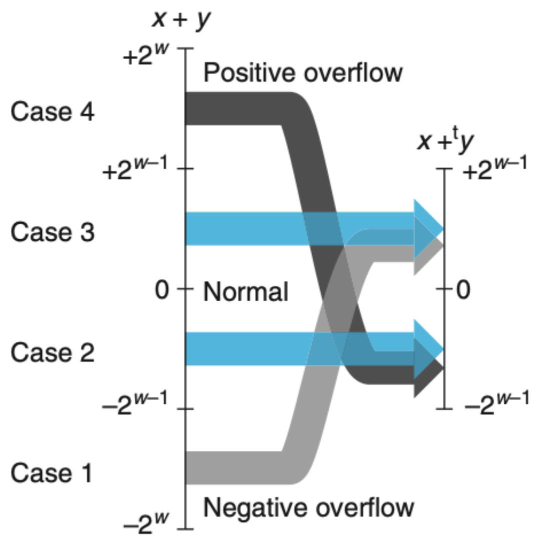
7: $-^u_4 x = 2^4 - 7 = 9$

A: $-^u_4 x = 2^4 - 10 = 6$

E: $-^u_4 x = 2^4 - 14 = 2$

2.3.2 Two's-Complement Addition

- Problem: 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.
  - Suppose: $-2^{w-1}(TMin_w) \leq x, y \leq 2^{w-1} - 1(TMax_w)$
  - Then we get, $-2^w \leq x + y \leq 2^w - 2$
  - Contradiction: two’s-complement addition with w bits can only represent from -2^{w-1} to $ (2^{w-1}-1)$, which there's some uncovered range.
- Two's-complement addition:
  - Suppose: $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$
    - $2^{w-1} - 1 = TMax_w$
    - $-2^{w-1} = TMin_w$
  - Formula:
    - Positive overflow (Case 4): if $2^w - 2 \geq x + y \geq 2^{w-1}$, $x +^t_w y = x + y - 2^w$
    - Normal (Case 2 and 3): If $-2^{w-1} \leq x + y \leq 2^{w-1} - 1$, $x +^t_w y = x + y$
    - Negative overflow (Case 1): if $-2^w \leq x + y < -2^{w-1}$, $x +^t_w y = x + y + 2^w$



- The w-bit two's-complement sum of two numbers has the exact same bit-level representation as the unsigned sum.
  - Most computers use the same machine instruction to perform either unsigned or signed addition - assembly add instruction.
- Derivation:
  - Define operation $+^t_w$:
    1. convert its argument to unsigned
    2. perform unsigned addition ($+^u_w$)
    3. convert back to two's complement
  - So: $x +^t_w y = U2T_w(T2U_w(x) +^u_w T2U_w(y))$

    because of formula: $T2U_w(x) = x_{w-1}2^w + x$

    = $U2T_w((x_{w-1}2^w + x) +^u_w (y_{w-1}2^w + y))$

    because of formula: $x +^u_w y = (x+y)mod2^w$

    = $U2T_w((x_{w-1}2^w + x + y_{w-1}2^w + y)mod2^w)$

    = $U2T_w((x+y)mod2^w)$
  - 4 cases:
    - Case 1: $-2^w \leq x+y < -2^{w-1}$
      - Then $(x+y)mod2^w = (x+y+2^w)mod2^w$
        - because $0 \leq x+y+2^w < 2^{w-1}$
        - $(x+y)mod2^w = x+y+2^w$
      - So $x +^t_w y = U2T_w((x+y)mod2^w) = U2T_w(x+y+2^w)$
      - because $0 \leq x+y+2^w < 2^{w-1}$
      - $x +^t_w y = x+y+2^w$
    - Case 2: $-2^{w-1} \leq x+y < 0$
      - Then $(x+y)mod2^w = (x+y+2^w)mod2^w = x+y+2^w$
      - So $x +^t_w y = U2T_w((x+y)mod2^w) = U2T_w(x+y+2^w)$
      - because $2^{w-1} \leq x+y+2^w < 2^w$ and if $x+y+2^w > TMax_w$, $U2T_w(x+y+2^w) = x+y+2^w-2^w$
      - $x +^t_w y = x+y+2^w-2^w = x+y$
    - Case 3: $2^{w-1} \geq x+y > 0$
      - Then $(x+y)mod2^w = x+y$
      - So $x +^t_w y = U2T_w((x+y)mod2^w) = U2T_w(x+y)$
      - because $2^{w-1} \geq x+y > 0$
      - $x +^t_w y = x+y$
    - Case 4: $2^{w-1} \leq x+y < 2^w$
      - Then $(x+y)mod2^w = x+y$
      - So $x +^t_w y = U2T_w((x+y)mod2^w) = U2T_w(x+y)$
      - because $2^{w-1} \leq x+y < 2^w$, $x+y > TMax_w$

        - $x +_w^t y = x + y - 2^w$
    - Example:

x	y	$x+y$	$x +_4^t y$	Case
−8	−5	−13	3	1
[1000]	[1011]	[10011]	[0011]	
−8	−8	−16	0	1
[1000]	[1000]	[10000]	[0000]	
−8	5	−3	−3	2
[1000]	[0101]	[11101]	[1101]	
2	5	7	7	3
[0010]	[0101]	[00111]	[0111]	
5	5	10	−6	4
[0101]	[0101]	[01010]	[1010]	

        - negative overflow yields a result 16 more than the integer sum, and positive overflow yields a result 16 less
        - **the result can be obtained by performing binary addition of the operands and truncating the result to 4 bits**
- Overflow Detection:
    - Principle: detect overflow in two's-complement addition
        - Assume: $TMin_w \leq x, y \leq TMax_w, s = x +_w^t y$
        - If x>0,y>0 but s<=0, POSITIVE OVERFLOW
        - If x<0,y<0 but s>=0, NEGATIVE OVERFLOW

Practice Problem 2.29

Fill in the following table. Give the integer values of the 5-bit arguments, the values of both their integer and two's-complement sums, the bit-level representation of the two's-complement sum, and the case from the derivation of Equation 2.13.

x	y	$x+y$	$x +_5^t y$	Case
[10100]	[10001]			
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			

Solution:

x=[10100] y=[10001] x+y= [100101] x$+_5^t$y=[00101] Negative Overflow - Case 1

x= -12 y= -15 x+y=-27 x$+_5^t$y=5

x=[11000] y=[11000] x+y=[110000] x$+_5^t$y=[10000] Case 2

x= -8 y= -8 x+y=-16 x$+_5^t$y= -16

x=[10111] y=[01000] x+y=[11111] x$+_5^t$y=[11111] Case 2

x= -9 y= 8 x+y=-1 x$+_5^t$y=-1

x=[00010] y=[00101] x+y=[00111] x$+_5^t$y=[00111] Case 3

$x= 2\ y= 5\ x+y=7\ x+^t_5 y=7$

$x=[01100]\ y=[00100]\ x+y=[10000]\ x+^t_5 y=[10000]$ Positive Overflow – Case 4

$x= 12\ y= 4\ x+y=16\ x+^t_5 y=-32 + 16 = -16$

Practice Problem 2.30

Write a function with the following prototype:

```c
/* Determine whether arguments can be added without overflow */
int tadd_ok(int x, int y);
```

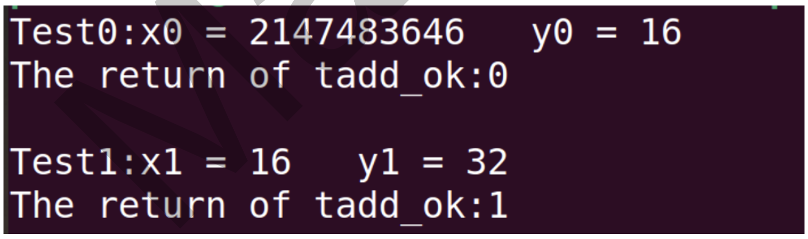
This function should return 1 if arguments x and y can be added without causing overflow.

```c
// P123.c
#include<stdio.h>
#include<limits.h>

int tadd_ok(int x, int y)
{
  int s = x+y;
  // If x>0,y>0 but s<=0, POSITIVE OVERFLOW
  if(x>0 && y>0 && s<=0) return 0;
        // If x<0,y<0 but s>=0, NEGATIVE OVERFLOW
        if(x<0 && y<0 && s>=0) return 0;
  return 1;
}

int main()
{
  int x0 = 0x7FFFFFFE;
  int y0 = 0x10;
  printf("Test0:");
  printf("x0 = %d\ty0 = %d\n",x0,y0);
  printf("The return of uadd_ok:%d\n\n",tadd_ok(x0,y0));

  int x1 = 0x10;
  int y1 = 0x20;
  printf("Test1:");
  printf("x1 = %d\ty1 = %d\n",x1,y1);
  printf("The return of uadd_ok:%d\n\n",tadd_ok(x1,y1));
  return 0;
}
```



```
Test0:x0 = 2147483646   y0 = 16
The return of tadd_ok:0

Test1:x1 = 16    y1 = 32
The return of tadd_ok:1
```

- &, |与&&, ||的区别:
  - &, |为位运算符;
    - &: 按位与操作;
    - |: 按位或操作;
  - &&, ||为逻辑运算符
    - &&: 如果两个操作数都非零，则表达式为真;
    - ||: 如果两个操作数中有任意一个非零，则表达式为真;

Practice Problem 2.31

Your coworker gets impatient with your analysis of the overflow conditions for two's-complement addition and presents you with the following implementation of tadd_ok:

```c
/* Determine whether arguments can be added without overflow */
/* WARNING: This code is buggy. */
int tadd_ok(int x, int y) {
        int sum = x+y;
  return (sum-x == y) && (sum-y == x);
}
```

You look at the code and laugh. Explain why.

Solution:

because x+y-x will always equal to y. It's abelian group.

Practice Problem 2.32

You are assigned the task of writing code for a function tsub_ok, with arguments x and y, that will return 1 if computing x-y does not cause overflow. Having just written the code for Problem 2.30, you write the following:

```c
/* Determine whether arguments can be subtracted without overflow */
/* WARNING: This code is buggy. */
int tsub_ok(int x, int y) {
    return tadd_ok(x, -y);
}
```

For what values of x and y will this function give incorrect results? Writing a correct version of this function is left as an exercise..

Solution:

we know $-2^{w-1} \le x < 2^{w-1}, \$ - 2^{31} \le x \le 2^{31} - 1$ which is not symmetric.

So if $y = -2^{w-1}$, then -y cannot be 2^{w-1} which is a positive overflow to $y = -2^{w-1}$ again.

Therefore, will cause an error.

2.3.3 Two's-Complement Negation 有符号非

- Suppose $TMin_w \le x \le TMax_w$, then its additive inverse (加法逆元) $-^t_w x$ under $+^t_w$ is:
  - if $x = TMin_w$, $-^t_w x = TMin_w$
  - if $x > TMin_w$, $-^t_w x = -x$
- Derivation:
  - If $x = TMin_w$, $x +^t_w x = TMin_w +^t_w TMin_w = -2^{w-1} - 2^{w-1} + 2^w = 0$ because of negative overflow.
  - if $x > TMin_w$, $x +^t_w (-x)$ will have no overflow ($= x + (-x)$) then equal to 0.

Practice Problem 2.33

We can represent a bit pattern of length w = 4 with a single hex digit. For a two's-complement interpretation of these digits, fill in the following table to determine the additive inverses of the digits shown:

![[./2_3.assets/Screenshot 2023-12-05 at 16.15.58.png" alt="Screenshot 2023-12-05 at 16.15.58" style="zoom:50%;" />

What do you observe about the bit patterns generated by two's-complement and unsigned negation?

Solution:

x: hex 2 [0010] decimal 2

$-^t_4 x$: hex E [1110] decimal -2

x: hex 3 [0011] decimal 3

$-^t_4 x$: hex D [1101] decimal -3

x: hex 9 [1001] decimal -7

$-^t_4 x$: hex 7 [0111] decimal 7

x: hex B [1011] decimal -5

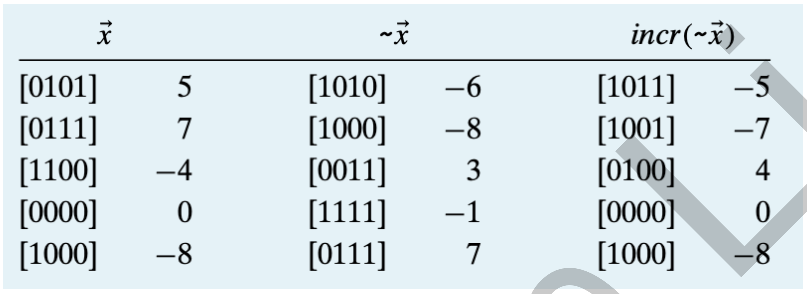
$-^t_4 x$: hex 5 [0101] decimal 5

x: hex C [1100] decimal -4

$-^t_4 x$: hex 4 [0100] decimal 4

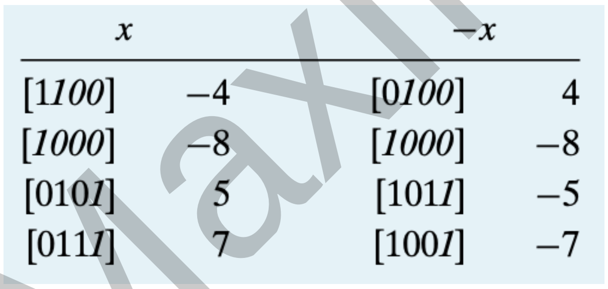
Bit-level representation of two's-complement negation

- 2 techniques to determine the two's-complement negation:
    - Technique 1: complement the bits (~x) and increment the result (~x+1)



$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
[0101]	5	[1010]	−6	[1011]	−5
[0111]	7	[1000]	−8	[1001]	−7
[1100]	−4	[0011]	3	[0100]	4
[0000]	0	[1111]	−1	[0000]	0
[1000]	−8	[0111]	7	[1000]	−8

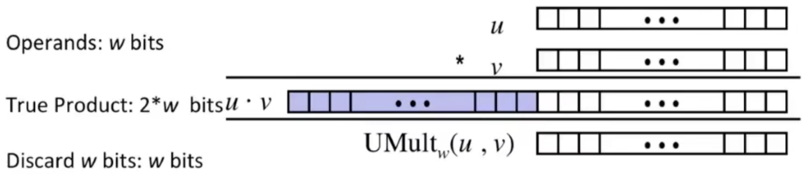
- Technique 2: perform two's-complement negation of a number x is based on splitting the bit vector into two parts.
    - Suppose k is the position of the rightmost 1 in $\vec{x}$
    - So original $\vec{x}$: $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$
    - Then negation: $[-x_{w-1}, -x_{w-2}, \ldots, -x_{k+1}, 1, 0, \ldots, 0]$



x		$-x$	
[1100]	−4	[0100]	4
[1000]	−8	[1000]	−8
[0101]	5	[1011]	−5
[0111]	7	[1001]	−7

2.3.4 Unsigned Multiplication

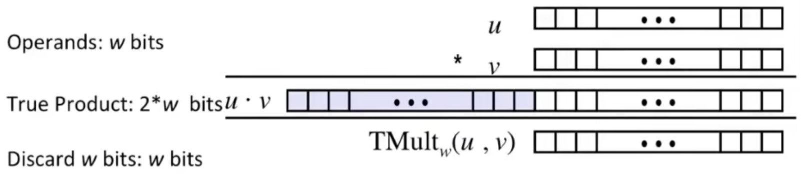
- Some truths:
    - Suppose: $0 \le x, y \le 2^w - 1$
    - Then $0 \le x \cdot y \le (2^w - 1)^2$
    - So the result requires 2w bits to represent



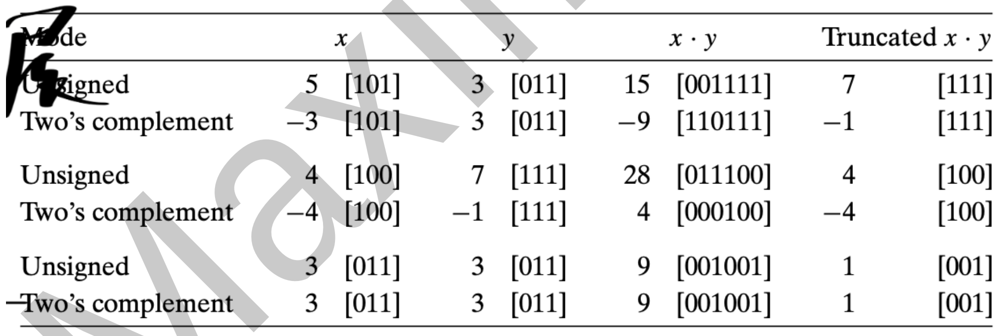
- Unsigned multiplication in C is defined to yield the w-bit value given by the low-order w bits of the 2w-bit integer product.
- Formula:
  - Suppose: $0 \le x, y \le UMax_w$
  - $x *^u_w y = (x \cdot y) mod 2^w$

2.3.5 Two's-Complement Multiplication

- Some truths:
  - Suppose: $-2^{w-1} \le x, y \le 2^{w-1} - 1$
  - Then $-2^{w-1} \cdot (2^{w-1} - 1) \le x \cdot y \le 2^{2w-2}$
  - So the result requires 2w bits to represent



- Signed multiplication ($x *^t_w y$) in C generally is performed by truncating the 2w-bit product to w bits:
  - computing product value modulo 2^w ($mod 2^w$)
  - converting from unsigned to two's complement ($U2T_w$)
- Formula1 - Two's-complement multiplication:
  - Suppose: $TMin_w \le x, y \le TMax_w$
  - Then: $x *^t_w y = U2T_w((x \cdot y) mod 2^w)$
- Fomula2 - Bit-level equivalence of unsigned and two's-complement multiplication
  - Suppose: $\vec{x}$ and $\vec{y}$ are bit vectors of length w.
  - Then we define: $x = B2T_w(\vec{x})$, $y = B2T_w(\vec{y})$
  - We also define: $x' = B2U_w(\vec{x})$, $y' = B2U_w(\vec{y})$
  - Conclusion: $T2B_w(x *^t_w y) = U2B_w(x' *^u_w y')$
- Example:



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]

```
Unsigned: x = [101],    y = [011]
Multiple steps:
                        [000101]
x                       [000011]
                         000101
                        000101
=                        001111


Two's complement: x = [101],    y = [011]
Multiple steps:
                        [111101]
x                       [000011]
                         111101
```

```
                  111101
                [110111]
```

- Perform both unsigned and two’s-complement multiplication, yielding 6-bit products, and then truncate these to 3 bits.
- The bit- level representations of both truncated products are identical for both unsigned and two’s-complement multiplication, even though the full 6-bit representations differ.

- Derivation:
  - $x' = B2U_w(\vec{x})$, $x = B2T_w(\vec{x})$, $y' = B2U_w(\vec{y})$, $y = B2T_w(\vec{y})$
  - We know: $x' = x + x_{w-1}2^w$ and $y' = y + y_{w-1}2^w$
  - Then:

    $x' *^u_w y' = (x' \cdot y') mod 2^w$

    $= ((x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)) mod 2^w$

    $= (x \cdot y) mod 2^w$
  - So:

$U2B_w(x' *^u_w y') = U2B_w((x \cdot y) mod 2^w)$

$= U2B_w(T2U_w(x *^t_w y))$

$= T2B_w(x *^t_w y)$

Practice Problem 2.34

Fill in the following table showing the results of multiplying different 3-bit numbers,

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned		[100]		[101]				
Two’s complement		[100]		[101]				
Unsigned		[010]		[111]				
Two’s complement		[010]		[111]				
Unsigned		[110]		[110]				
Two’s complement		[110]		[110]				

x=[100] y=[101]

Unsigned: x=4 y=5 $x \cdot y$=$[00010100]_2$ 20_{10} Truncated $x \cdot y$= $[100]_2$ 4_{10}

```
     [000100]
  x  [000101]
      000100
    000100
 =  00010100
```

Two's complement: x=-4 y=-3 $x \cdot y$= $[001100]_2$ 12_{10} Truncated $x \cdot y$= $[100]_2$ -4_{10}

```
     [111100]
  x  [111101]
      111100
    111100
   111100
  111100
 111100
 =  [001100]
```

x=[010] y=[111]

Unsigned: x= 2 y= 7 $x \cdot y$= $[001110]_2$ 14_{10} Truncated $x \cdot y$= $[110]_2$ 6_{10}

```
       [000010]
   x   [000111]
        000010
       000010
      000010
  =     001110
```

Two's complement: x= 2 y= -1 $x \cdot y$= $[111110]_2$ -2_{10} Truncated $x \cdot y$= $[110]_2$ -2_{10}

```
       [000010]
   x   [111111]
        000010
       000010
      000010
     000010
    000010
   000010
  =     111110
```

x=[110] y=[110]

Unsigned: x= 6 y= 6 $x \cdot y$= $[100100]_2$ 36_{10} Truncated $x \cdot y$= $[100]_2$ 4_{10}

```
       [000110]
   x   [000110]
       000110
      000110
  =     100100
```

Two's complement: x= -2 y= -2 $x \cdot y$= $[000100]_2$ 4_{10} Truncated $x \cdot y$= $[100]_2$ 4_{10}

```
       [111110]
   x   [111110]
       111110
      111110
     111110
    111110
   111110
  =     000100
```

Practice Problem 2.35

You are given the assignment to develop code for a function tmult_ok that will determine whether two arguments can be multiplied without causing overflow. Here is your solution:

```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
    int p = x*y;
    /* Either x is zero, or dividing p by x gives y */
    return !x || p/x == y;
}
```

You test this code for a number of values of x and y, and it seems to work properly. Your coworker challenges you, saying, “If I can’t use subtraction to test whether addition has overflowed (see Problem 2.31), then how can you use division to test whether multiplication has overflowed?”

Devise a mathematical justification of your approach, along the following lines. First, argue that the case x = 0 is handled correctly. Otherwise, consider w-bit numbers x (x$\neq$ 0), y, p, and q, where p is the result of performing two’s-complement multiplication on x and y, and q is the result of dividing p by x.

1. Show that x·$y, theintegerproductofxandy, canbewrittenintheformx\cdot y = p + t$2^w, where t $\neq$ 0 if and only if the computation of p overflows.

Solution:

assume: u is the lower w bits of x·y

v is the upper w bits of x·y

then: x·y = v2^w + u

because: p = $x *_w^t y$ = $U2T_w(u)$

then: u = $T2U_w(p)$ = p + $p_{w-1}2^w$

so: x·y = v2^w + u = v2^w + p + $p_{w-1}2^w$ = p + (v+p_{w-1})2^w which t = v+p_{w-1}

2. Show that p can be written in the form p=x$\cdot $q+r,where|r|<|x|.

Solution:

We know that q is the result of dividing p by x.

p = x·q + r

3. Show that q = y if and only if r=t=0.

Solution:

assume: q = y

we know: t = v+p_{w-1} and

so: x·y = p + t2^w = x·q+r+ t2^w = x·y+r+ t2^w

Practice Problem 2.36

For the case where data type int has 32 bits, devise a version of tmult_ok (Problem 2.35) that uses the 64-bit precision of data type int64_t, without using division.

Solution:

```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
   int64_t x64 = (int64_t)x;
   int64_t y64 = (int64_t)y;
   int64_t p64 = x64*y64;
      int p = x*y;
   return p == p64;
}
```

- Example:

```
/* Illustration of code vulnerability similar to that found in
 * Sun’s XDR library.
 */
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size); // overflow
```

```
        if (result == NULL)
            /* malloc failed */
            return NULL;
        void *next = result;
        int i;
        for(i = 0; i < ele_cnt; i++)
        {
             /* Copy object i to destination */
                   memcpy(next, ele_src[i], ele_size);
                                              /* Move pointer to next memory region */
                   next += ele_size;
        }
        return result;
    }
```

- Code:

```
// P129.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>
#include<stdint.h>
 void* copy_elements(void *ele_src[], int ele_cnt, uint32_t ele_size) {
     /*
      * Allocate buffer for ele_cnt objects, each of ele_size bytes
      * and copy from locations designated by ele_src
      */
   // void *malloc(size_t size);
     void *result = malloc(ele_cnt * ele_size); // overflow
     printf("The value of ele_cnt*ele_size: %u",ele_cnt*ele_size);
     if (result == NULL)
         /* malloc failed */
         return NULL;
     void *next = result;
     int i;
     for(i = 0; i < ele_cnt; i++)
     {
          /* Copy object i to destination */
          memcpy(next, ele_src[i], ele_size);
          /* Move pointer to next memory region */
          next += ele_size;
     }
     return result;
 }

int main()
{
  int cnt = 1048577; // 2^20 + 1
  uint32_t size = 4096; // 2^12
  int test_array[0x100] = {0};
  void* res = copy_elements((void*)test_array, cnt, size);

  return 0;
}
```

Practice Problem 2.37

You are given the task of patching the vulnerability in the XDR code shown in the aside on page 136 for the case where both data types int and size_t are 32 bits. You decide to eliminate the possibility of the multiplication overflowing by computing the number of bytes to allocate using data type uint64_t. You replace the original call to malloc (line 9) as follows:

```
uint64_t asize =        ele_cnt * (uint64_t) ele_size;
```

```
void *result =          malloc(asize);
```

Recall that the argument to malloc has type size_t.
A. Does your code provide any improvement over the original?

Solution:

```
void *malloc(size_t size)
```

no improvements. On 32bits system, size_t is 32bits. So asize will be casted and the same overflow happens.

B. How would you change the code to eliminate the vulnerability?

Solution:

Add the code to check overflow.

```
uint64_t asize =        ele_cnt * (uint64_t) ele_size;
size_t st_size = asize;
if(asize != st_size) return NULL; // overflow
void *result =          malloc(asize);
```

2.3.6 Multiplying by Constants

- Problem:
  - The integer multiply instruction on many machines was fairly slow, requiring 10 or more clock cycles.
  - Other integer operations—such as addition, subtraction, bit-level operations, and shifting—required only 1 clock cycle.
- Idea: replace multiplications by constant factors with combinations of shift and addition operations.
- Formula - multiplication by a power of 2:
  - assume x = $[x_{w-1}, x_{w-2}, \ldots, x_0]$
  - Then: $x2^k$ = $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$ where k zeros have been added.
  - Derivation:

    $B2U_{w+k}([x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0])$

    $= \sum_{i=0}^{w-1} x_i 2^{i+k}$

    $= x2^k$
  - When shifting left by k for a fixed word size, the high-order k bits are discarded, yielding.

    $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$
- Formula - unsigned multiplication by a power of 2:
  - assume $0 \leq k < w$
  - then: $x << k = x *_w^u 2^k$
- Formula - two's-complement multiplication by a power of 2
  - assume $0 \leq k < w$
  - then: $x << k = x *_w^t 2^k$
- Multiplying by a power of 2 can cause overflow with either unsigned or two's-complement arithmetic.
  - Example: [1011] << 2 -> [101100] -> truncating to 4 bits: [1100]
- 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:

- Example (x*15):
    - We know: $15 = 2^3 + 2^2 + 2^1 + 2^0$
    - Then rewrite: x*15 = (x<<3) + (x<<2) + (x<<1) + (x<<0) - 3 shifts and 3 additions
    - Or: $15 = 2^4 - 2^0$
    - Then also rewrite: x*15 = (x<<4) - (x<<0) - 1 shift and 1 subtraction

Practice Problem 2.38

As we will see in Chapter 3, the lea instruction can perform computations of the form (a<<k)+b, where k is either 0, 1, 2, or 3, and b is either 0 or some program value. The compiler often uses this instruction to perform multiplications by constant factors. For example, we can compute 3*a as (a<<1) + a.

Considering cases where b is either 0 or equal to a, and all possible values of k, what multiples of a can be computed with a single lea instruction?

Extension:

2 kinds of assembly languages:

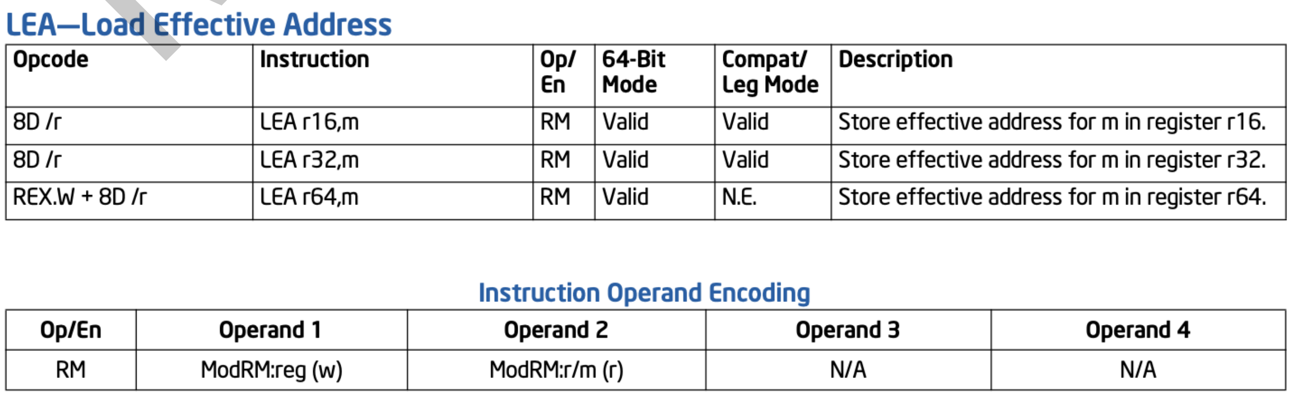
- x86
    - Intel - windows
    - At&t - linux: which csapp choose

  http://tuttlem.github.io/2014/03/25/assembly-syntax-intel-at-t.html

Intel Code	AT&T Code
`mov eax,1`	`movl $1,%eax`
`mov ebx,0ffh`	`movl $0xff,%ebx`
`int 80h`	`int $0x80`
`mov ebx, eax`	`movl %eax, %ebx`
`mov eax,[ecx]`	`movl (%ecx),%eax`
`mov eax,[ebx+3]`	`movl 3(%ebx),%eax`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`add eax,[ebx+ecx*2h]`	`addl (%ebx,%ecx,0x2),%eax`
`lea eax,[ebx+ecx]`	`leal (%ebx,%ecx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

- Arm

// Intel® 64 and IA-32 Architectures Software Developer's Manuals:



LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for m in register r16.
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for m in register r32.
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for m in register r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description - Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand).

Solution:

2^k and 2^k+1

1,2,3,4,5,8,9

- Example - x*K
  - Condition: K is some constant a nd can be expressed in binary: $[(0\ldots0)(1\ldots1)\ldots]$
  - Suppose: 30 = [00011110]
  - Then: n = 4, m = 1
  - Therefore:
    - Form A: (x<<n) + (x<<(n-1)) + ... + (x<<m)
    - Form B: (x<<(n+1)) - (x<<m)
- Most compilers only perform this optimization when a small number of shifts, adds, and subtractions suffice.

Practice Problem 2.39

How could we modify the expression for form B for the case where bit position n is the most significant bit?

Solution:

Form B: (x<<(n+1)) - (x<<m)

Since n is the MSB, w = n+1

(x<<(n+1)) - (x<<m) = (x<<w)-(x<<m) = -(x<<m)

Practice Problem 2.40

For each of the following values of K, find ways to express x * K using only the specified number of operations, where we consider both additions and subtractions to have comparable cost. You may need to use some tricks beyond the simple form A and B rules we have considered so far.

K	Shifts	Add/Subs	Expression
7	1	1	
30	4	3	
28	2	1	
55	2	2	

$x \times 7 \ = \ x \times (2^3-1) = x << 3 - x$

$x \times 30 \ = \ x \times (16+8+4+2) \ = \ x << 4 + x << 3 + x << 2 + x << 1$

$x \times 28 \ = \ x \times (32-4) \ = \ x << 5 - x << 2$

$x \times 55 \ = \ x \times (64-8-1) \ = \ x << 6 - x << 3 - x$

Practice Problem 2.41

For a run of ones starting at bit position n down to bit position m (n ≥ m), we saw that we can generate two forms of code, A and B. How should the compiler decide which form to use?

Solution:

- if n == m,
  - form A: 1 shift
  - form B: 2 shifts and 1 sub

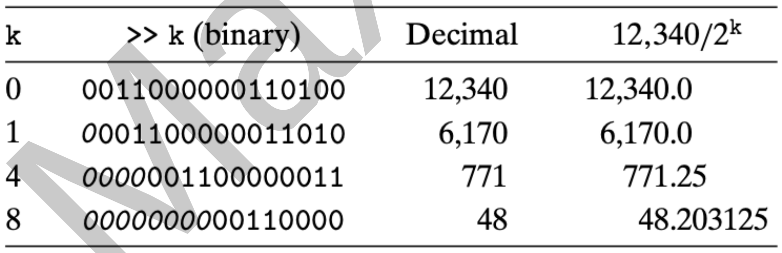
    - compiler decision: A
- if n == m + 1
    - form A: 2 shifts and 1 add
    - form B: 2 shifts and 1 sub
    - compiler decision: A or B
- if n > m + 1
    - form A: >2 shifts and >1 adds
    - form B: 2 shifts and 1 sub
    - compiler decision: B

2.3.7 Dividing by Powers of 2

- Problem: Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles.
    - Solution: Dividing by a power of 2 can 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.
- Integer division (整数除法) always rounds toward zero.
    - Basic1: For any real number a, a' = $\lfloor a \rfloor$ (向下取整), then a' $\leq$ a < a' + 1
        - Example: $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$
    - Basics2: For any real number a, a' = $\lceil a \rceil$ (向上取整), then a' - 1 < a $\leq$ a'
        - Example: $\lceil 3.14 \rceil = 4$, $\lceil -3.14 \rceil = -3$
    - Statement1: for x$\geq$ \$$0 and y > 0, the result of integer division is$\$$\lfloor x/y \rfloor$
    - Statement2: for x<0 and y>0, or x>0 and y<0, the result of integer division is $\lceil x/y \rceil$
    - To summarize, integer division should **round down a positive result but round up a negative one.**

Unsigned division by a power of 2

- Formula:
    - Assuming: unsigned variables x and k, and $0 \leq k < w$
    - Then: **x>>k =** $\lfloor x/2^k \rfloor$
- Example - dividing unsigned numbers by powers of 2:



k	>> k (binary)	Decimal	$12{,}340/2^k$
0	0011000000110100	12,340	12,340.0
1	0001100000011010	6,170	6,170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

    - A logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.
- Derivation:
    - Assume:
        - unsigned integer x = $[x_{w-1}, x_{w-2}, \ldots, x_0]$ and $0 \leq k < w$
        - unsigned integer x' = $[x_{w-1}, x_{w-2}, \ldots, x_k]$
        - unsigned integer x'' = $[x_{k-1}, \ldots, x_0]$
    - Then: $x = 2^k x' + x''$
    - And: $0 \leq x'' < 2^k$, $\lfloor x/2^k \rfloor = x'$
    - We also know: a logical right shift of $[x_{w-1}, x_{w-2}, \ldots, x_0]$ by k (x>>k) is:

      $[0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots, x_k]$ which is x'

- Therefore: x>>k = $\lfloor x/2^k \rfloor$

Two’s-complement division by a power of 2

- Complex than unsigned division: arithmetic right shift.

Rounding down

- Assume:
  - Two's-complement value $x \geq 0$ and unsigned value k ($0 \leq k < w$)
  - Then: $x >> k = \lfloor x/2^k \rfloor$
- If $x \geq 0$, 0 is the MSB, so the effect of an arithmetic shift is the same as for a logical right shift, which is $x >> k = \lfloor x/2^k \rfloor$
- Derivation:
  - Assume:
    - Two's-complement integer $x = [x_{w-1}, x_{w-2}, \ldots, x_0]$
    - $0 \leq k < w$
    - Two's-complement integer $x' = [x_{w-1}, x_{w-2}, \ldots, x_k]$
    - Unsigned integer $x'' = [x_{k-1}, \ldots, x_0]$
    - Then: $x = 2^k x' + x''$
    - We know: $0 \leq x'' < 2^k$, then $0 \leq x''/2^k < 1$
    - $x/2^k$ = $(2^k x' + x'')/2^k$ = $x' + x''/2^k$
    - So: $x' = \lfloor x/2^k \rfloor$
    - We also know:
      - after right shifting by k (x>>k) on $[x_{w-1}, x_{w-2}, \ldots, x_0]$
      - we get: x>>k = $[x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_k]$ (arithmetically shift) = $[x_{w-1}, x_{w-2}, \ldots, x_k]$ = x' = $\lfloor x/2^k \rfloor$

Rounding up - correct improper rounding by "biasing" (偏置)

- The improper rounding that occurs when a negative number is shifted right by “biasing” the value before shifting.

k	>> k (binary)	Decimal	$-12{,}340/2^k$
0	1100111111001100	−12,340	−12,340.0
1	*1*110011111100110	−6,170	−6,170.0
4	*1111*110011111100	−772	−771.25
8	*11111111*11001111	−49	−48.203125

  - when k = 4, -12340>>4 = -772 while $-12340/2^4 = -771.25 \approx -771$ but $\neq -772$
  - this improper rounding cause the wrong difference!
- Assume:
  - Two's-complement value x<0 and unsigned value k ($0 \leq k < w$)
  - Then: $(x + (1 << k) - 1) >> k = \lceil x/2^k \rceil$ while biasing is (1<<k)-1
- Example:

k	Bias	−12,340 + bias (binary)	>> k (binary)	Decimal	$-12{,}340/2^k$
0	0	1100111111001100	1100111111001100	−12,340	−12,340.0
1	1	110011111100110*1*	*1*110011111100110	−6,170	−6,170.0
4	15	110011111101*1011*	*1111*110011111101	−771	−771.25
8	255	11010000*11001011*	*11111111*11010000	−48	−48.203125

    - biasing = (1<<k)-1
    - When k = 0, biasing = 0 - no rounding required, no bits affected
    - When k = 1, biasing = 1 - no rounding is required, adding the bias only affects bits that are shifted off
        - 0b 1100111111001100 + 0b 1 = 0b 1100111111001101, >>1
    - When k = 4, biasing = 15 - rounding is required, adding the bias causes the upper bits to be incremented, so that the result will be rounded toward zero.
        - 0b 1100111111001100 + 0b 1111 = 0b1100111111011011, /2^4 = -771.25
    - When k = 8, biasing = 255 - rounding is required, adding the bias causes the upper bits to be incremented, so that the result will be rounded toward zero.
        - 0b 1100111111001100 + 0b 1111 1111 = 0b 1101000011001011,/2^8 = -48.203125
- Derivation:
    - We know the property: $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$
    - Then: $x + (1 << k) - 1 = x + 2^k - 1$
    - Then: $(x + (1 << k) - 1) >> k = (x + 2^k - 1) >> k = \lfloor (x + 2^k - 1)/2^k \rfloor = \lceil x/2^k \rceil$

Formula combining rounding up and down

- C expression:

$x/2^k = (x < 0 ? x + (1 << k) - 1 : x) >> k$

Practice Problem 2.42

Write a function div16 that returns the value x/16 for integer argument x. Your function should not use division, modulus, multiplication, any conditionals (if or ?:), any comparison operators (e.g., <, >, or ==), or any loops. You may assume that data type int is 32 bits long and uses a two's-complement representation, and that right shifts are performed arithmetically.

Solution:

```
int div16(int x)
{
  // (x<0? x+(1<<4)-1 : x)>>4
  int signed_bit = x >> 31;
  int bias = signed_bit & 0xF;
  return (x + bias)>>4;
}
```

Sample:

```
// P136.c
#include <stdio.h>

int div16(int x)
{
  // (x<0? x+(1<<4)-1 : x)>>4
  int signed_bit = x >> 31;
  int bias = signed_bit & 0xF;
  return (x + bias)>>4;
}

int main()
{
  int x0 = 16, x1 = 31, x2 = 33;

  printf("%d/16=%d\n",x0,div16(x0));
  printf("%d/16=%d\n",x1,div16(x1));
  printf("%d/16=%d\n",x2,div16(x2));
```

```
    return 0;
}
```

Practice Problem 2.43

In the following code, we have omitted the definitions of constants M and N:

```
#define M      /* Mystery number 1 */
#define N      /* Mystery number 2 */
int arith(int x, int y) {
    int result = 0;
    result = x*M + y/N; /* M and N are mystery numbers. */
    return result;
}
```

We compiled this code for particular values of M and N. The compiler optimized the multiplication and division using the methods we have discussed. The following is a translation of the generated machine code back into C:

```
/* Translation of assembly code for arith */
int optarith(int x, int y) {
    int t = x;
    x <<= 5;
    x -= t;
    if (y < 0) y += 7;
    y >>= 3;  /* Arithmetic shift */
    return x+y;
}
```

What are the values of M and N?

Solution:

M value:

We can get M value from x<<=5 and x-=t,

Thus: $x = x << 5 - x = x \times 2^5 - x = x \times (2^5 - 1) = x \times 31$

So: M = 31

N value:

from if (y<0) y+=7, means 7 = bias = (1<<k)-1 = (1<<3)-1

$N = 2^3 = 8$

2.3.8 Final Thoughts on Integer Arithmetic

- We have seen that some of the conventions in the C language can yield some surprising results, and these can be sources of bugs that are hard to recognize or understand.

Practice Problem 2.44

Assume data type int is 32 bits long and uses a two's-complement representation for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```
    int x = foo();   /* Arbitrary value */
    int y = bar();   /* Arbitrary value */
    unsigned ux = x;
    unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all values of x and y, or (2) give values of x and y for which it is false (evaluates to 0):

1. (x>0)||(x-1<0)

   FALSE. If x = $TMin_{32}$, then x - 1 = $TMax_{32}$ > 0
2. (x&7)!=7||(x<<29<0)

   Solution:

   If(((x&7)!=7) == 0), then (x&7)==7, so $x = [x_{31}, x_{30}, \ldots, x_3, 1, 1, 1]$

   Then x<<29 = $[1, 1, 1, 0, \ldots, 0]$ < 0 TRUE
3. (x*x)>=0

   FALSE. If x = 0xFFFF, then x*x = 0xFFFE0001 < 0
4. x<0 || -x<=0

   TRUE.
5. x>0 || -x>=0

   FALSE, If x = $TMin_{32}$, then -x = $TMax_{32} + 1$ = $TMin_{32}$
6. x+y == uy+ux

   TRUE. Same bit representation for signed and unsigned numbers.