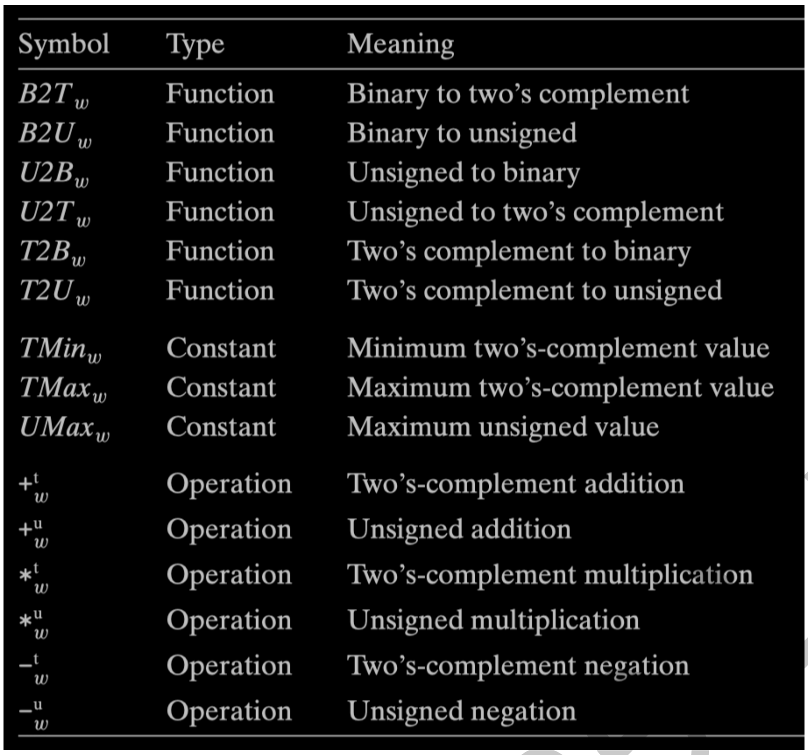
2_2 Integral Data Types

- 2 different ways that bits can be used to encode integers:
  - one that only represent nonnegative numbers.
  - one that can represent negative, zero, and positive numbers.
- Mathematical terminology we introduce to precisely define and characterize how computers encode and operate on integer data.



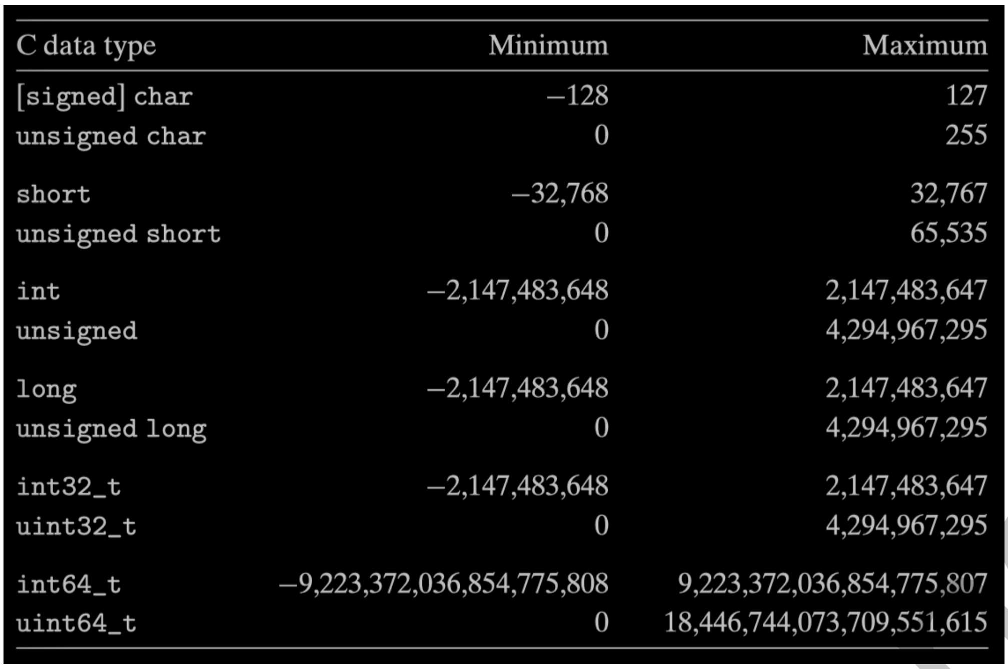
Symbol	Type	Meaning
$B2T_w$	Function	Binary to two's complement
$B2U_w$	Function	Binary to unsigned
$U2B_w$	Function	Unsigned to binary
$U2T_w$	Function	Unsigned to two's complement
$T2B_w$	Function	Two's complement to binary
$T2U_w$	Function	Two's complement to unsigned
$TMin_w$	Constant	Minimum two's-complement value
$TMax_w$	Constant	Maximum two's-complement value
$UMax_w$	Constant	Maximum unsigned value
$+^{t}_{w}$	Operation	Two's-complement addition
$+^{u}_{w}$	Operation	Unsigned addition
$*^{t}_{w}$	Operation	Two's-complement multiplication
$*^{u}_{w}$	Operation	Unsigned multiplication
$-^{t}_{w}$	Operation	Two's-complement negation
$-^{u}_{w}$	Operation	Unsigned negation

- can invert mappings:
  - $U2B(x) = B2U^{-1}(x)$
  - $T2B(x) = B2T^{-1}(x)$

2.2.1 Integal Data Types

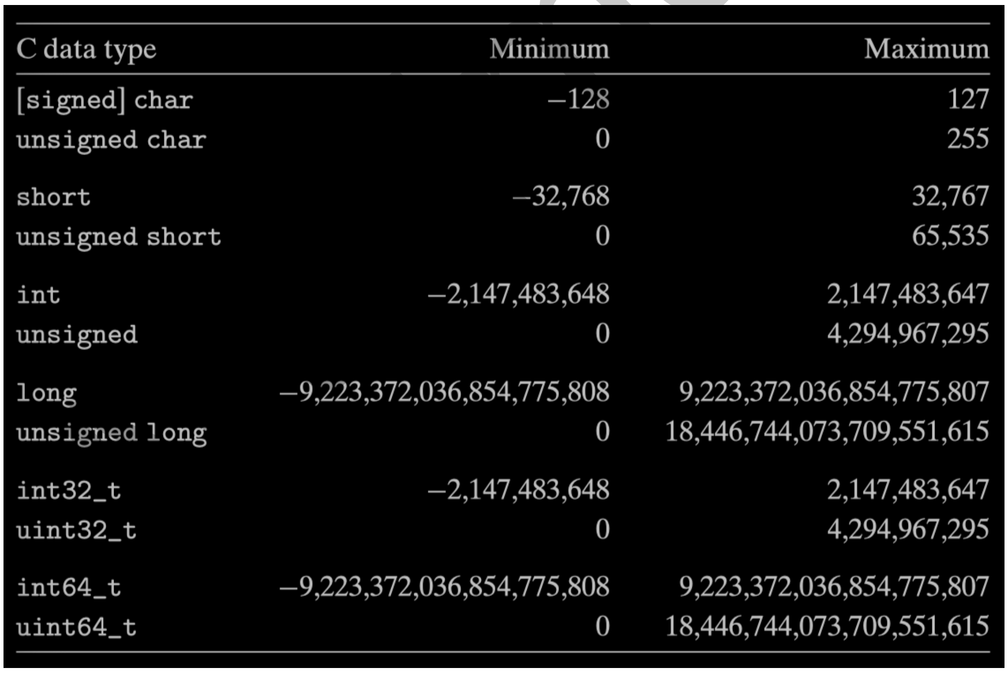
- C supports a variety of integral data types, which represent finite ranges of integers.

    - Typical 32bit program:



C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

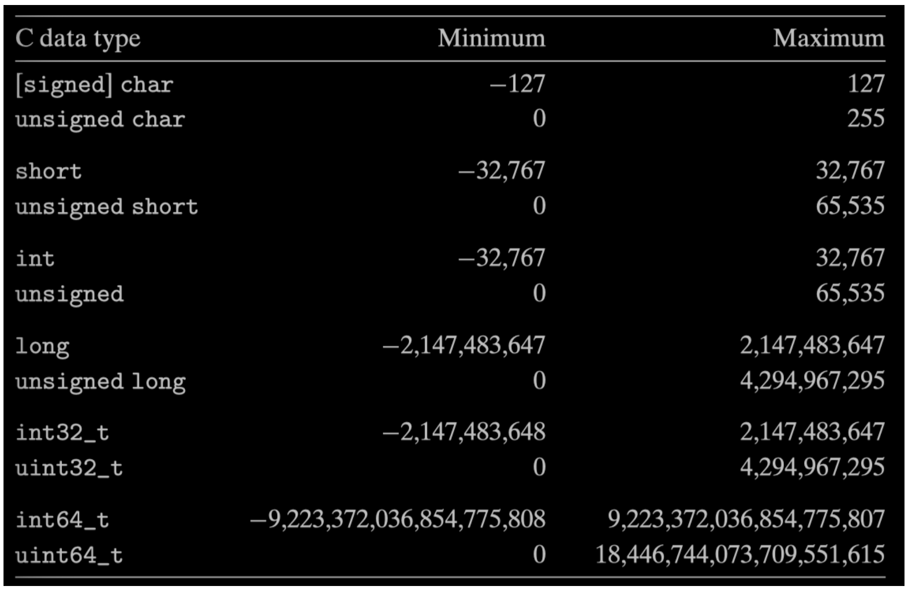
- char:
    - min - [1000 0000] = -128
    - max - [0111 1111] = 127
- unsigned char:
    - min - 0
    - max - [1111 1111] = 255
- short
    - min - [1000 0000 0000 0000] = -32768
    - max - [0111 1111 1111 1111] = 32767
- Typical 64bit program:



C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

- Each type can specify a size with keyword **char**, **short**, **long**, as well as an indication of whether the represented numbers are all **nonnegative** (declared as unsigned), or possibly **negative** (the default.)
- Based on the byte allocations, the different sizes allow different ranges of values to be represented. The only machine-dependent range indicated is for size designator long.
- Most 64-bit programs use an 8-byte representation, giving a much wider range of values than the 4-byte representation used with 32-bit programs.

- The ranges are not symmetric.
- The C standards define minimum ranges of values that each data type must be able to represent.

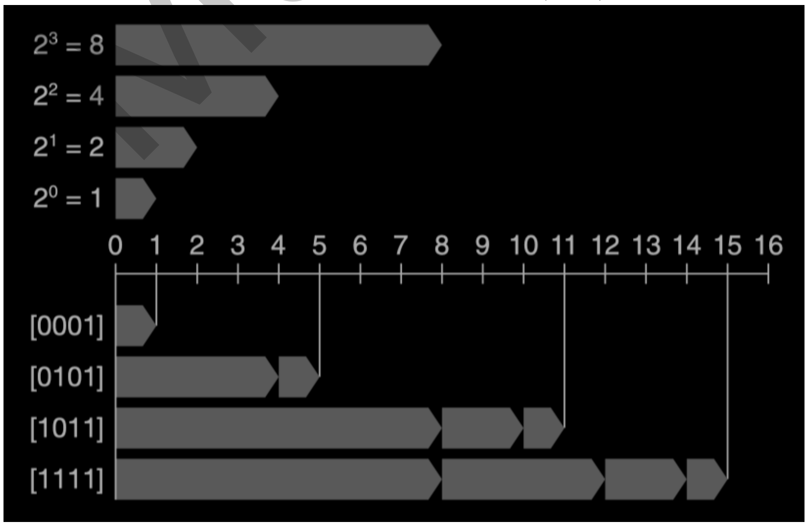


C data type	Minimum	Maximum
[signed] char	−127	127
unsigned char	0	255
short	−32,767	32,767
unsigned short	0	65,535
int	−32,767	32,767
unsigned	0	65,535
long	−2,147,483,647	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

  - Their ranges are the same or smaller than the typical implementations.
  - With the exception of the fixed-size data types, we see that they require only a symmetric range of positive and negative numbers.
  - Data type int could be only implemented with 2-byte numbers.
  - The size long can be implemented with 4-byte numbers, and it typically is for 32-bit programs.
  - The fixed-size data types (int32_t, uint32_t, int64_t and uint64_t) guarantee that the ranges of values will be exactly those given by the typical numbers, including the asymmetry between negative and positive.

2.2.2 Unsigned Encodings (无符号编码)

- An integer data type of w bits:
  - treat as $\vec{x}$(向量) or $[x_{w-1}, x_{w-2}, \ldots, x_0]$
  - Treating x as a number written in binary notation, we obtain the unsigned interpretation of x.
  - Each bit x i has value 0 or 1, with the latter case indicating that value 2^i should be included as part of the numeric value.
  - $B2U_w$ (Binary to Unsigned): $B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i$
- Mapping from bit vectors to integers (w=4):

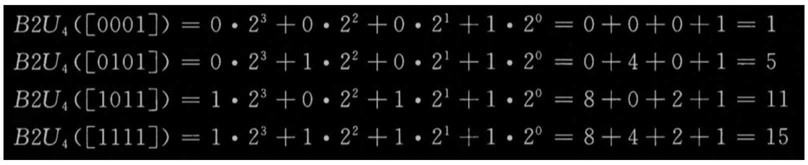


[0001] = $1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3$ = 1

[0101] = $1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3$ = 5

[1011] = $1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3$ = 11

- Examples:



$$B2U_4([0001]) = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1$$
$$B2U_4([0101]) = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5$$
$$B2U_4([1011]) = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11$$
$$B2U_4([1111]) = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15$$

- The range of values that can be represented using w bits:
  - The least value is given by bit vector [00 . . . 0], which is integer value 0.
  - The greatest value is given by bit vector [11 . . . 1], which is:
    $UMax_w = \sum_{i=0}^{w-1} 2^i = 2^w - 1$
  - Example:
    $UMax_4 = B2U_4([1111]) = 2^4 - 1 = 15$
  - The function $B2U_w$ can be defined as a mapping(映射) $B2U_w : \{0,1\}^w \rightarrow \{0, \ldots, UMax_w\}$.
  - What is $\{0,1\}^w$?
    - P123 of 《Discrete Mathematics and Its Applications 7th Edition - Rosen》
    - The Cartesian product of the sets A_1, A_2 , . . . , A_n , denoted by A_1× A_2 × · · · × A_n , is the set of ordered n-tuples (a_1 , a_2 , . . . , a_n), where a_i belongs to A_i for i = 1, 2, . . . , n. In other words,
      $A_1 \times A_2 \times A_3 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i \quad for \quad i = 1, 2, \ldots, n\}$
    - So
      - $\{0,1\}^2$ = $\{0,1\} \times \{0,1\}$ = {(0,0), (0,1), (1,0), (1,1)}
      - $\{0,1\}^3$ = {(0,0,0), (0,0,1), (0,1,0), (0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)}
- The unsigned binary representation has the important property that every number between 0 and $2^w - 1$ has a unique encoding as a w-bit value.

```
Principle: Uniqueness of unsigned encoding - 无符号数编码的唯一性
```

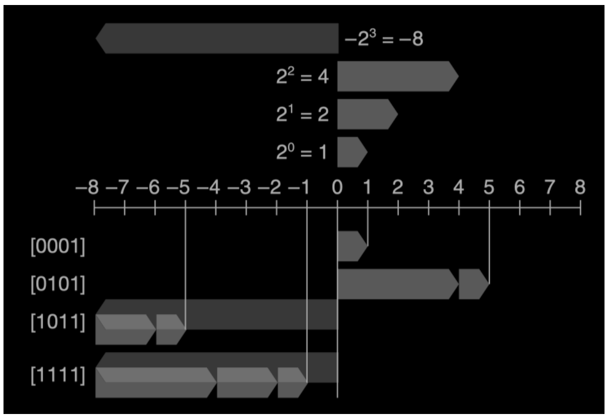
- Function $B2U_w$ is a bijection （双射）
  - what is bijection? - a function f that goes two ways: it maps a value x to a value y where y = f (x), but it can also operate in reverse.
  - For every y, there is a unique value x such that f (x) = y.
  - For every x, there is a unique value y such that $x = f^{-1}(y)$, while f^{-1} is the inverse function.
- The function B2U maps each bit vector of length w to a unique number between 0 and $2^w - 1$, and it has an inverse, which we call $U2B_w$ (for “unsigned to binary”), that maps each number in the range 0 to $2^w - 1$ to a unique pattern of w bits.

2.2.3 Two's-Complement Encodings (有符号编码)

- The most common computer representation of signed numbers is known as two’s-complement form.
  - This is defined by interpreting the most significant bit (MSB - 最高有效位) of the word to have negative weight.
  - MSB: 0 for non-negative and 1 for negative.
- $B2T_w$ - Binary To Two's complement for length w
- Definition:
  - For vector $\overrightarrow{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:
    $B2T_w(\overrightarrow{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
  - The most significant bit (MSB) x_{w-1} is also called the sign bit.

- When the sign bit is set to 1, the represented value is negative.
- When set to 0, the value is nonnegative.
- Example - $B2T_4$

$$\begin{array}{lclclcl}
B2T_4([0001]) & = & -0\cdot 2^3+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0 & = & 0+0+0+1 & = & 1 \\
B2T_4([0101]) & = & -0\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0 & = & 0+4+0+1 & = & 5 \\
B2T_4([1011]) & = & -1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0 & = & -8+0+2+1 & = & -5 \\
B2T_4([1111]) & = & -1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0 & = & -8+4+2+1 & = & -1
\end{array}$$



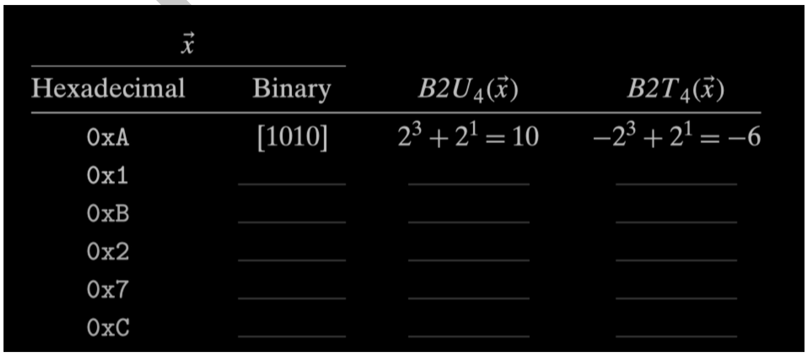
- The value range of a w-bit two's complement number.
  - $TMin_w$ = -2^{w-1}
  - $TMax_w = \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$
  - Example (w=4):
    - $TMin_4 = B2T_4([1000]) = -2^3 = -8$
    - $TMax_4 = B2T_4([0111]) = 2^2 + 2^1 + 2^0 = 7$
  - $B2T_w$ is a mapping of bit patterns of length w to numbers between $TMin_w$ and $TMax_w$,: $B2T_w : \{0,1\}^w \rightarrow \{TMin_w, \ldots, TMax_w\}$
- Every number within the representable range has a unique encoding as a w-bit two’s-complement number.

```
Principal: uniqueness of two’s-complement encoding - 补码/有符号编码的唯一性
```

- Function $B2T_w$ is a bijection（双射）.
  - $T2B_w$ is the inverse of $B2T_w$
  - For a number x, such that $TMin_w \leq x \leq TMax_w$, $T2B_w(x)$ is the (unique) w-bit pattern that encodes x.

Practice Problem 2.17

Assuming w = 4, we can assign a numeric value to each possible hexadecimal digit, assuming either an unsigned or a two’s-complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of 2:



$\vec{x}$			
Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
0xA	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
0x1	____	____	____
0xB	____	____	____
0x2	____	____	____
0x7	____	____	____
0xC	____	____	____

Solution:

Hex Binary $B2U_4(\overrightarrow{x})$ $B2T_4(\overrightarrow{x})$

`0x1 [0001]` $1 \times 2^0 = 1$ $1 \times 2^0 = 1$

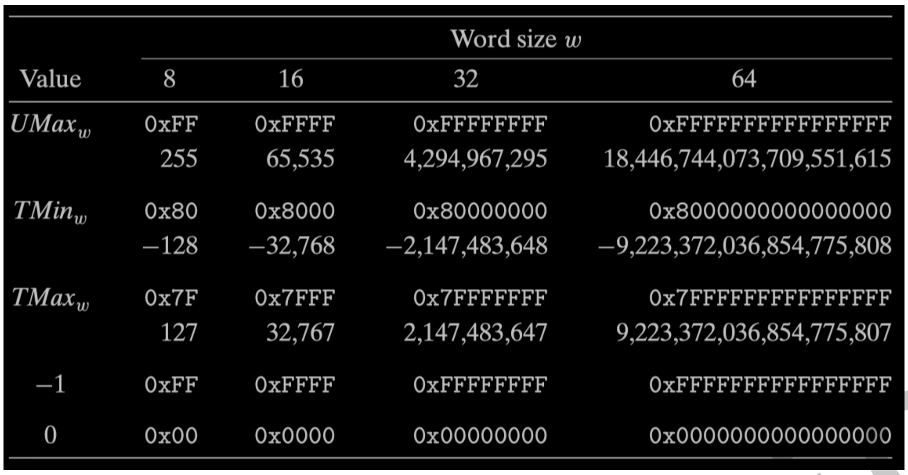
`0xB [1011]` $2^3 + 2^1 + 2^0$=11 $-2^3 + 2^1 + 2^0$= -5

`0x2 [0010]` 2^1 = 2 2^1 = 2

`0x7 [0111]` $2^2 + 2^1 + 2^0$=7 $2^2 + 2^1 + 2^0$=7

`0xC [1100]` $2^3 + 2^2$=12 $-2^3 + 2^2 = -4$

- The bit patterns and numeric values for several important numbers for different word sizes (w).



	Word size w			
Value	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65,535	0xFFFFFFFF 4,294,967,295	0xFFFFFFFFFFFFFFFF 18,446,744,073,709,551,615
$TMin_w$	0x80 −128	0x8000 −32,768	0x80000000 −2,147,483,648	0x8000000000000000 −9,223,372,036,854,775,808
$TMax_w$	0x7F 127	0x7FFF 32,767	0x7FFFFFFF 2,147,483,647	0x7FFFFFFFFFFFFFFF 9,223,372,036,854,775,807
−1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

  - 0xFF = [1111 1111] = $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ = -1
  - |TMin| = |TMax| + 1: |[1000...0]| = |[0111...1]| + 1
  - UMax = 2*TMax + 1: [0111...1]*2 + 1= [1111...10] + 1 = [1111...1] = UMax
  - -1 has the same bit representation as UMax—a string of all ones.
  - Numeric value 0 is represented as a string of all 0 in both representations.
- The file <limits.h> in the C library defines a set of constants delimiting the ranges of the different integer data types for the particular machine on which the compiler is running.

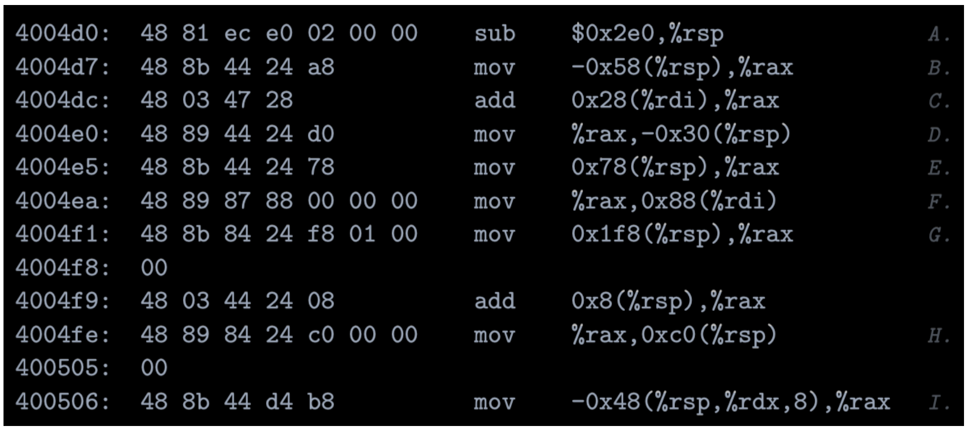
```
INT_MAX
INT_MIN
UINT_MAX
```

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

int main()
{
        printf("INT_MAX value is: %x\n",INT_MAX);       // TMax_w
        printf("INT_MIN value is: %x\n",INT_MIN);       // TMin_w
        printf("UINT_MAX value is: %x\n",UINT_MAX);     //UMax_w

        return 0;
}
```

Practice Problem 2.18

In Chapter 3, we will look at listings generated by a disassembler, a program that converts an executable program file back to a more readable ASCII form. These files contain many hexadecimal numbers, typically representing values in two's complement form. Being able to recognize these numbers and understand their significance (for example, whether they are negative or positive) is an important skill.

For the lines labeled A–I (on the right) in the following listing, convert the hexadecimal values (in 32-bit two's-complement form) shown to the right of the instruction names (sub, mov, and add) into their decimal equivalents:

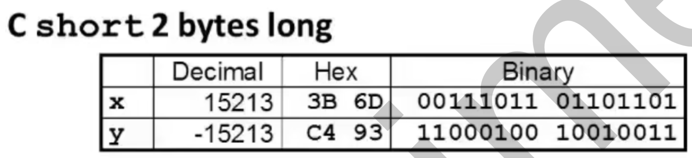


```
4004d0:  48 81 ec e0 02 00 00    sub    $0x2e0,%rsp              A.
4004d7:  48 8b 44 24 a8          mov    -0x58(%rsp),%rax         B.
4004dc:  48 03 47 28             add    0x28(%rdi),%rax          C.
4004e0:  48 89 44 24 d0          mov    %rax,-0x30(%rsp)         D.
4004e5:  48 8b 44 24 78          mov    0x78(%rsp),%rax          E.
4004ea:  48 89 87 88 00 00 00    mov    %rax,0x88(%rdi)          F.
4004f1:  48 8b 84 24 f8 01 00    mov    0x1f8(%rsp),%rax         G.
4004f8:  00
4004f9:  48 03 44 24 08          add    0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00    mov    %rax,0xc0(%rsp)          H.
400505:  00
400506:  48 8b 44 d4 b8          mov    -0x48(%rsp,%rdx,8),%rax  I.
```

Solution:

```
A. 0x2e0 = 2*16*16 + 14 * 16 = 736
B. -0x58 = -(5*16 + 8) = -88
C. 0x28 = 2*16 + 8 = 40
D. -0x30 = -(3*16 + 0) = -48
E. 0x78 = 7*16 + 8 = 120
F. 0x88 = 8*16 + 8 = 136
G. 0x1f8 = 1*16*16 + 15*16 + 8 = 760
H. 0xc0 = 12*16 + 0 = 192
I. -0x48 = -(4*16 + 8) = -72
```

2.2.4 Conversions between Signed and Unsigned



C short 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- C allows casting between different numeric data types.
  - Example:
    - Suppose variable x is declared as int and u as unsigned.

```
int x;
unsigned int u;
```

- The expression $(unsigned)x$ converts the value of x to an unsigned value, and $(int)u$ converts the value of u to a signed integer.

```
(unsigned)x
(int)u
```

- Converting a negative value to unsigned might yield zero. - FALSE

```
// Conversion.c
int x0 = -100;
unsigned int y0 = (unsigned int)x0;
printf("%d", y0);
```

        - Converting an unsigned value that is too large to be represented in two’s complement form might yield TMax. – FALSE

```c
// Conversion.c
unsigned int x1 = 0xFFFFFFDD;
int y1 = (int)x1;
printf("%d",y1);
```

        - The effect of casting is to **keep the bit values identical but change how these bits are interpreted.**

```c
// P99
short int v = -12345;
unsigned short uv = (unsigned short)v;
printf("v = %d, uv = %u\n", v, uv);
```

        - Output:

```
v = -12345, uv = 53191
0xCFC7
= 53191 if type is unsigned int
= -12345 if type is int
```

```c
// P99
unsigned u = 4294967295u; // UMax
int tu = (int)u;
printf("u = %u, tu = %d\n", u, tu);
```

        - Output:

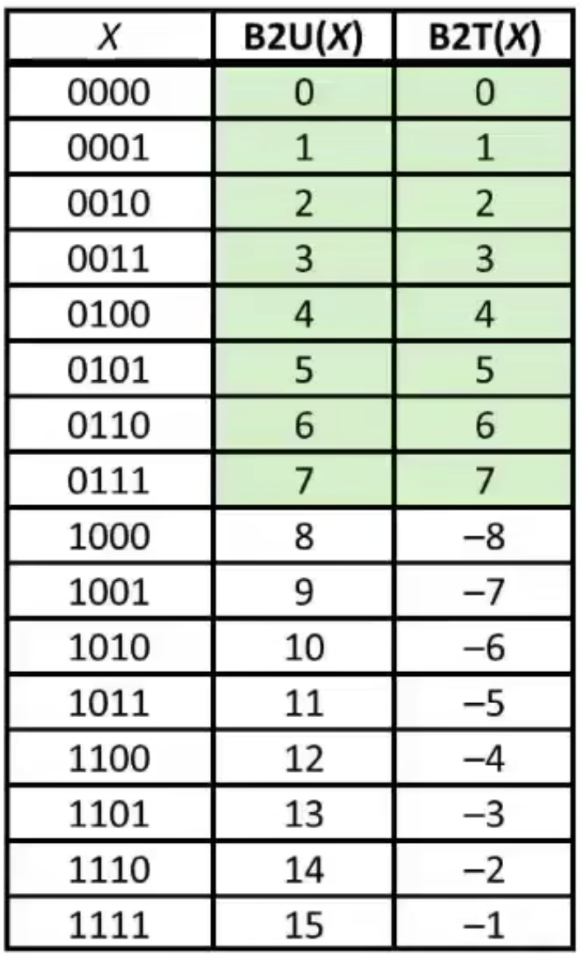
```
u = 4294967295, tu = -1
```

- This is a general rule for how most C implementations handle conversions between signed and unsigned numbers with the same word size—**the numeric values might change, but the bit patterns do not.**
    - $T2B_w$ – convert two's-complement to bit representations;
        - For $T2B_w(x)$, $TMin_w \leq x \leq TMax_w$
    - $U2B_w$ – convert unsigned to bit representation;
        - For $U2B_w(x)$, $0 \leq x \leq UMax_w$
    - So $T2U_w(x) = B2U_w(T2B_w(x))$
    - $T2U_{16}(-12345) = B2U_{16}(T2B_{16}(-12345)) = 53191$

```
-12345 = 0xCFC7 = 53191
```

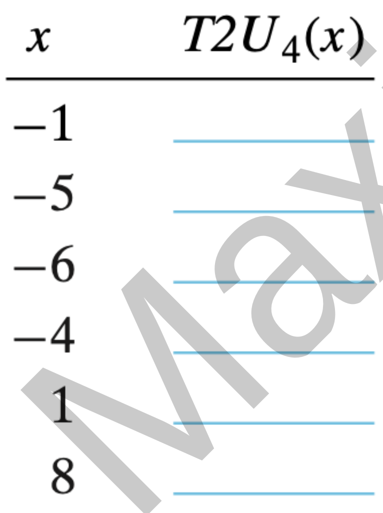
    - $U2T_w(x) = B2T_w(U2B_w(x))$
    - $U2T_{32}(4294967295) = B2T_{32}(U2B_{32}(4294967295)) = -1$

```
4292967295 = 0x FFFF FFFF = -1
```



X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	−8
1001	9	−7
1010	10	−6
1011	11	−5
1100	12	−4
1101	13	−3
1110	14	−2
1111	15	−1

Practice Problem 2.19



x	$T2U_4(x)$
-1	
-5	
-6	
-4	
1	
8	

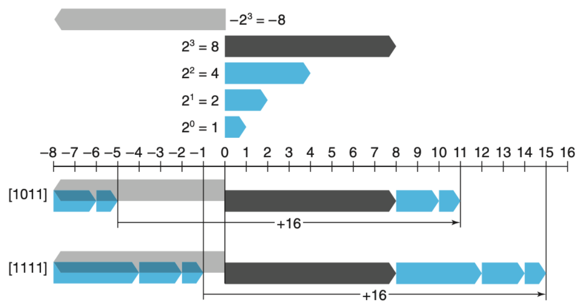
Solution: $T2U_w(x) = B2U_w(T2B_w(x))$

```
w = 4:
-1 = 0b 1111 = 15
-5 = 0b 1011 = 11
-6 = 0b 1010 = 10
-4 = 0b 1100 = 12
 1 = 0b 0001 = 1
 8
```

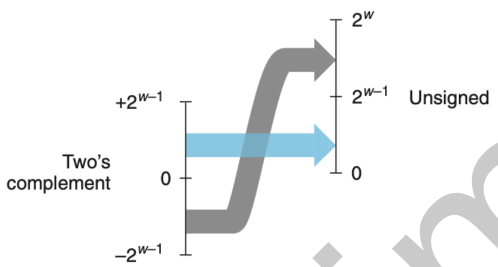
- Conversion from two's complement to unsigned:

- For $TMin_w \le x \le TMax_w$:
  - if x < 0, $T2U_w(x) = x + 2^w$
  - If x >= 0, $T2U_w(x) = x$
- Example: $T2U_{16}(-12345) = -12345 + 2^{16} = 53191$

- derivation:
  - we know: $B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i = x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
  - we also know: $B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
  - Then we get: $B2U_w(\vec{x}) = B2T_w(\vec{x}) + x_{w-1}2^w$
  - Therefore:
    $T2U_w(x) = B2U_w(T2B_w(x)) = B2T_w(T2B_w(x)) + x_{w-1}2^w = x + x_{w-1}2^w$
- Comparing unsigned and two's-complement representations for w = 4. The weight of the most significant bit is -8 for two's complement and +8 for unsigned, yielding a net difference of 16.



- Conversion from two's complement to unsigned. Function *T2U* converts negative numbers to large positive numbers.



- When mapping a signed number to its unsigned counterpart, **negative numbers are converted to large positive numbers**, while nonnegative numbers remain unchanged.

Practice Problem 2.20

Explain how Equation 2.5 applies to the entries in the table you generated when solving Problem 2.19.

x	$T2U_4(x)$
−1	
−5	
−6	
−4	
1	
8	

- Equation 2.5:
  - For $TMin_w \le x \le TMax_w$:
    - If x < 0, $T2U_w(x) = x + 2^w$
    - If x >= 0, $T2U_w(x) = x$

Solution:

w = 4: TMax = 7, TMin = -8

-1 = -1 + 2^4 = 15

$-5 = -5 + 2^4$ = 11

$-6 = -6 + 2^4 = 10$

$-4 = -4 + 2^4 = 12$

1 = 1

8

- Conversion from unsigned to two's complement:
  - For $0 \le u \le UMax_w$:
    - If $u \le TMax_w$, $U2T_w(u) = u$
    - If $u > TMax_w$, $U2T_w(u) = u - 2^w$
  - For small (≤ $TMax_w$) numbers, the conversion from unsigned to signed preserves the numeric value. Large (> $TMax_w$) numbers are converted to negative values.
- Derivation:
  - we know: $B2U_w(\vec{u}) = \sum_{i=0}^{w-1} u_i 2^i = u_{w-1}2^{w-1} + \sum_{i=0}^{w-2} u_i 2^i$
  - we also know: $B2T_w(\vec{u}) = -u_{w-1}2^{w-1} + \sum_{i=0}^{w-2} u_i 2^i$
  - Then we get: $B2T_w(\vec{u}) = B2U_w(\vec{u}) - u_{w-1}2^w$
  - Therefore: $U2T_w(u) = B2T_w(U2B_w(u)) = B2U_w(U2B_w(u)) - u_{w-1}2^w = u - u_{w-1}2^w$

2.2.5 Signed versus Unsigned in C

- C supports both signed and unsigned arithmetic for all of its integer data types.
  - Most numbers are signed by default.
  - Declaring a constant such as 12345 or 0x1A2B, the value is considered signed.
  - Adding character 'U' or 'u' as a suffix creates an unsigned constant; for example, 12345U or 0x1A2Bu.
- C allows conversion between unsigned and signed, but the underlying bit representation does not change.
  - Explicit casting（显式强制转换）:

```c
int tx, ty;
unsigned ux, uy;

tx = (int)ux;
uy = (unsigned)ty;
```

  - Implicit casting（隐式强制转换）:

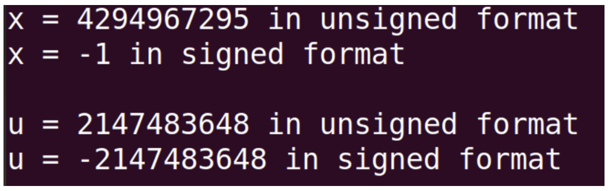
```c
int tx, ty;
unsigned ux, uy;

tx = ux;
uy = ty;
```

  - directives (%d %u and %x - 格式控制字符) of printf function to cast:

```c
// P104
#include<stdio.h>
int main()
{
  int x = -1;
  unsigned u = 2147483648;

  printf("x = %u in unsigned format\n x = %d in signed format\n\n", x, x);
```

```
    printf("u = %u in unsigned format\nu = %d in signed format\n\n", u, u);

    return 0;
}
```

- $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$
- $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$



```
x = 4294967295 in unsigned format
x = -1 in signed format

u = 2147483648 in unsigned format
u = -2147483648 in signed format
```

- Some possibly nonintuitive behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities.

Expression			Type	Evaluation
0	==	0U	Unsigned	1
-1	<	0	Signed	1
-1	<	0U	Unsigned	0 *
2147483647	>	-2147483647-1	Signed	1
2147483647U	>	-2147483647-1	Unsigned	0 *
2147483647	>	(int) 2147483648U	Signed	1 *
-1	>	-2	Signed	1
(unsigned) -1	>	-2	Unsigned	1

- When no type casting in expression, treat expression as signed;
- When there's signed type casting (i, signed etc) in expression, treat expression as signed;
- When there's unsigned type casting (u, unsigned etc) in expression, treat expression as unsigned;
- When there are a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned.

Practice Problem 2.21

Assuming the expressions are evaluated when executing a 32-bit program on a machine that uses two's-complement arithmetic, fill in the following table describing the effect of casting and relational operations:

Expression	Type	Evaluation
-2147483647-1 == 2147483648U		
-2147483647-1 < 2147483647		
-2147483647-1U < 2147483647		
-2147483647-1 < -2147483647		
-2147483647-1U < -2147483647		

Solution:

```
-2147483647-1 == 2147483648U -> unsigned -> 1
-2147483647-1 < 2147483647 -> signed -> 1
-2147483647-1U < 2147483647 -> unsigned -> 0
-2147483647-1 < -2147483647 -> signed -> 1
-2147483647-1U < -2147483647 -> unsigned -> 1
```

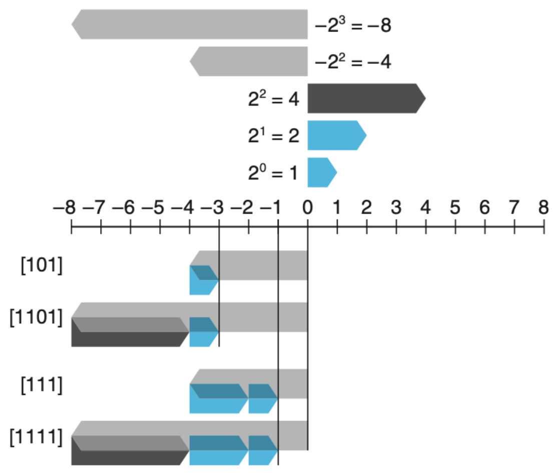
2.2.6 Expanding the Bit Representation of a Number

- One common operation is to convert between integers having different word sizes while retaining the same numeric value.
- 2 kinds of expanding the bit representation:
  - Zero extension (零扩展): to convert an **unsigned number** to a larger data type by simply adding leading zeros to the representation:
    - Define $\vec{u} = [u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w
    - and $\vec{u}' = [0, \ldots, 0, u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w' where w' > w
  - Then $B2U_w(\vec{u}) = B2U_{w'}(\vec{u}')$
- Signed extension (符号扩展): to convert a **two's-complement** number to a larger data type by **adding copies of the most significant bit**(MSB) to the representation:
  - Define $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w
  - and $\vec{x}' = [x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w' where w' > w
  - Then $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$
- Example - verify zero extension and signed extension:

```c
        // P106.c
#include<stdio.h>

typedef unsigned char* byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
        int i;
        for(i = 0; i < len; i++) printf(" %.2x", start[i]);
        printf("\n");
}

int main()
{
  short sx = -12345;
  unsigned short usx = sx;
  int x = sx;                                  // signed extension
  unsigned ux = usx;    // zero extension

  /* signed extension */
  printf("Before signed extension: sx = %d -\t",sx);
  show_bytes((byte_pointer)&sx, sizeof(short));
  printf("After signed extension: x = %d -\t",x);
  show_bytes((byte_pointer)&x, sizeof(int));
  printf("\n");

  /* zero extension */
  printf("Before zero extension: usx = %d -\t",usx);
  show_bytes((byte_pointer)&usx, sizeof(short));
  printf("After zero extension: ux = %u -\t",ux);
  show_bytes((byte_pointer)&ux, sizeof(int));

  return 0;
}
```

```
Before signed extension: sx = -12345      c7 cf
After signed extension: x = -12345        c7 cf ff ff

Before zero extension: usx = 53191        c7 cf
After zero extension: ux = 53191          c7 cf 00 00
```

- Another example of signed extension - w=3 extend to w=4



- `[101] extend to [1101]`
  - `[101] = -4 + 1 = -3`
  - `[1101] = -8 + 4 + 1 = -3`
- `[111] extend to [1111]`
  - `[111] = -4 + 2 + 1 = -1`
  - `[1111] = -8 + 4 + 2 + 1 = -1`
- The process of proving signed extension:
  - We want to prove:
    $B2T_{w+k}([x_{w-1},\dots,x_{w-1},x_{w-1},x_{w-2},\dots,x_0]) = B2T_w([x_{w-1},x_{w-2},\dots,x_0])$
  - We only need to prove:
    $B2T_{w+1}([x_{w-1},x_{w-1},x_{w-2},\dots,x_0]) = B2T_w([x_{w-1},x_{w-2},\dots,x_0])$
  - Then:
    $B2T_{w+1}([x_{w-1},x_{w-1},x_{w-2},\dots,x_0])$
    $= -x_{w-1}2^{w+1-1} + \sum_{i=0}^{w-1} x_i 2^i$
    $= -x_{w-1}2^{w} + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
    $= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
    $= B2T_w([x_{w-1},x_{w-2},\dots,x_0])$
  - Therefore, we reach the conclusion.

Practice Problem 2.22

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.

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program. Consider the following code:

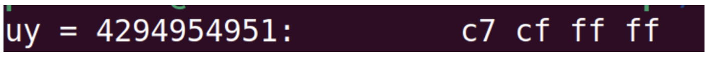
```c
// P109.c
short sx = -12345;
unsigned uy = sx;

printf("uy = %u:\t", uy);
show_bytes((byte_pointer)&uy, sizeof(unsigned));
```

When run on a big-endian machine, this code causes the following output to be printed:

```
uy = 4294954951:  ff ff cf c7
```

This shows that, when converting from short to unsigned, the program first changes the size and then the type. That is, (unsigned) sx is equivalent to (unsigned) (int) sx, evaluating to 4,294,954,951, not (unsigned) (unsigned short) sx, which evaluates to 53,191. Indeed, this convention is required by the C standards.

Solution:



```
uy = 4294954951:          c7 cf ff ff
```

This problem is to find out how our machine cast from short to unsigned:

By logically, there are 2 paths:

- short(-12345) -> unsigned short (53191)-> unsigned (53191)
  - The result will be: 53191 0x c7 cf 00 00
- short (-12345)-> int (-12345) -> unsigned (4294954951) -- Our machine choose this path!!!
  - The result will be: 4294954951 0x c7 cf ff ff

In Summary, **when converting from short to unsigned, the program first changes the size and then the type.**

Practice Problem 2.23

Consider the following C functions:

```c
int fun1(unsigned word){
        return (int)((word<<24)>>24);
}

int fun2(unsigned word){
  return ((int)word<<24)>>24;
}
```

Assume these are executed as a 32-bit program on a machine that uses two's- complement arithmetic. Assume also that right shifts of signed values are performed arithmetically, while right shifts of unsigned values are performed logically.

A. Fill in the following table showing the effect of these functions for several example arguments. You will find it more convenient to work with a hexadecimal representation. Just remember that hex digits 8 through F have their most significant bits equal to 1.

w	fun1(w)	fun2(w)
0x00000076		
0x87654321		
0x000000C9		
0xEDCBA987		

B. Describe in words the useful computation each of these functions performs.

Solutions:

A.

```
0x 00000076
        fun1(w): <<24  - 0x76000000 -> >>24 - 0x00000076 -> (int) - 0x00000076
        fun2(w): (int) - 0x00000076 -> <<24 - 0x76000000 -> >>24  - 0x00000076
0x 87654321
        fun1(w): <<24  - 0x21000000 -> >>24 - 0x00000021 -> (int) - 0x00000021
        fun2(w): (int) - 0x87654321 -> <<24 - 0x21000000 -> >>24  - 0x00000021
0x 000000C9
        fun1(w): <<24  - 0xC9000000 -> >>24 - 0x000000C9 -> (int) - 0x000000C9
        fun2(w): (int) - 0x000000C9 -> <<24 - 0xC9000000 -> >>24  - 0xFFFFFFC9
```

```
0x EDCBA987
        fun1(w): <<24  - 0x87000000 -> >>24 - 0x00000087 -> (int) - 0x00000087
        fun2(w): (int) - 0xEDCBA987 -> <<24 - 0x87000000 -> >>24  - 0xFFFFFF87
```

B.
fun1: extract the low 8 bits of the argument (w) and do a zero extension to 32 bits. Return the signed integer.

fun2: extract the low 8 bits of the argument (w) and do a signed extension to 32 bits. Return the signed integer.

2.2.7 Truncating Numbers

- For both unsigned and signed, bits are truncated but the result should be reinterpreted.
- Example:

```
int x = 53191;
short sx = (short)x;
int y = sx;
```

Truncation of an unsigned number

- Truncation of an **unsigned number**:
  - Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$
  - Let $\vec{x}'$ be result of truncating it to k bits $[x_{k-1}, x_{k-2}, \ldots, x_0]$
  - Let $x = B2U_w(\vec{x})$ and $x' = B2U_k(\vec{x}')$
  - Therefore: $x' = x mod 2^k$
- Proving:
  $x mod 2^k = B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) mod 2^k$

  $= [\sum_{i=0}^{w-1} x_i 2^i] mod 2^k$

  $= (x_{w-1}2^{w-1} + x_{w-2}2^{w-2} + \ldots + x_k 2^k + x_{k-1}2^{k-1} + x_{k-2}2^{k-2} + \ldots + x_0 2^0) mod 2^k$

  $= [\sum_{i=0}^{k-1} x_i 2^i] mod 2^k$

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

  $= B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0])$

  $= x'$

Truncation of a two's-complement number

- Truncation of a **two's-complement number**:
  - Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$
  - Let $\vec{x}'$ be result of truncating it to k bits $[x_{k-1}, x_{k-2}, \ldots, x_0]$
  - Let $x = B2T_w(\vec{x})$ and $x' = B2T_k(\vec{x}')$
  - Therefore: $x' = U2T_k(x mod 2^k)$
- Proving:
  $U2T_k(x mod 2^k) = U2T_k(B2U_k(\vec{x}'))$
  $= B2T_k(\vec{x}') = x'$

Practice Problem 2.24

Suppose we truncate a 4-bit value (represented by hex digits 0 through F) to a 3- bit value (represented as hex digits 0 through 7.) Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's- complement interpretations of those bit patterns.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
1	1	1		1	
3	3	3		3	
5	5	5		5	
C	4	12		−4	
E	6	14		−2	

Solution:

Unsigned truncation: $x' = xmod2^k$

Signed truncation: $x' = U2T_k(xmod2^k)$

```
w = 4, k = 3, 2^k = 8
Unsigned:
1  Truncated: 1
3  Truncated: 3
5  Truncated: 5
12 Truncated: 4
14 Truncated: 6

Two's complement:
1  Truncated: 0b 001 -> 1
3  Truncated: 0b 011 -> 3
5  Truncated: 0b 101 -> -3
-4 0b1100 Truncated: 0b100 -> -4
-2 0b1110 Truncated: 0b110 -> -2
```

2.2.8 Advice on Signed versus Unsigned

- The implicit casting of signed to unsigned leads to some non-intuitive behavior. Nonintuitive features often lead to program bugs, and ones involving the nuances of implicit casting can be especially difficult to see.

Practice Problem 2.25

Consider the following code that attempts to sum the elements of an array a, where the number of elements is given by parameter length:

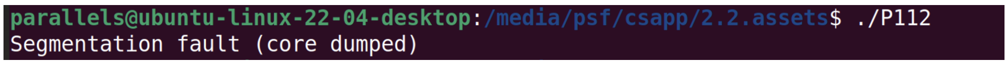
```c
float sum_elements(float a[], unsigned length)
{
        int i;
  float result = 0;

  for(i = 0; i <= length-1; i++) result += a[i];

  return result;
}
```

When run with argument length equal to 0, this code should return 0. Instead, it encounters a memory error. Explain why this happens. Show how this code can be corrected.

Solution:

```c
// P112.c
#include<stdio.h>

```

```c
float sum_elements(float a[], unsigned length)
{
        int i;
  float result = 0;

  for(i = 0; i <= length-1; i++) result += a[i];

  return result;
}

int main()
{
  float s[2] = {0,1};
  float sum = sum_elements(s, 0);

  return 0;
}
```



```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2.2.assets$ ./P112
Segmentation fault (core dumped)
```

Correction: i <= length-1 to i<length

Practice Problem 2.26

You are given the assignment of writing a function that determines whether one string is longer than another. You decide to make use of the string library function strlen having the following declaration:

```c
size_t strlen(const char *s);

int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}
```

When you test this on some sample data, things do not seem to work quite right. You investigate further and determine that, when compiled as a 32-bit program, data type size_t is defined (via typedef) in header file stdio.h to be unsigned.

A. For what cases will this function produce an incorrect result?

B. Explain how this incorrect result comes about.

C. Show how to fix the code so that it will work reliably.

Solution:

A.

strlen(s) < strlen(t)

B.

strlen returns unsigned integer, which makes the value of strlen(s) - strlen(t) is never negative.

C.

return strlen(s) > strlen(t)