C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	–	–	16
pointer	4	8	8

- X **&** Y   Intersection
- X **|** Y   Union
- X **^** Y   Symmetric difference
- **~**Y   Complement

- **Examples (char data type)**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Numeric Ranges

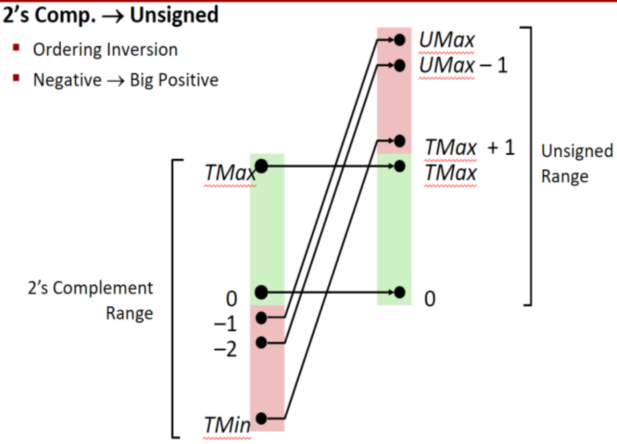
- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Conversion Visualized

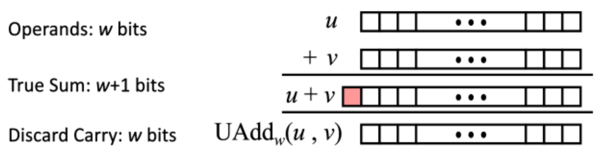
- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations <, >, ==, <=, >=
  - Examples for $W = 32$: **TMIN = -2147483648 ,** **TMAX = 2147483647**

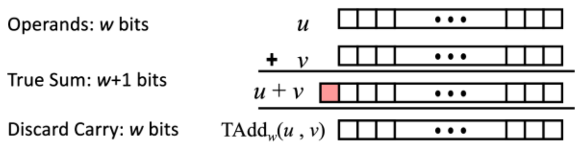
Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Unsigned Addition



- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**
  $s = \text{UAdd}_w(u, v) = u + v \mod 2^w$

Two’s Complement Addition (for Signed)

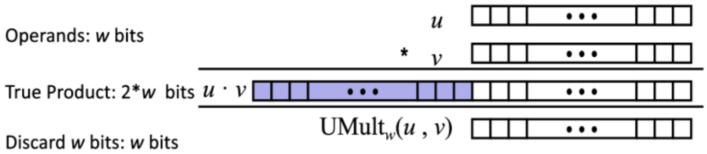


- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

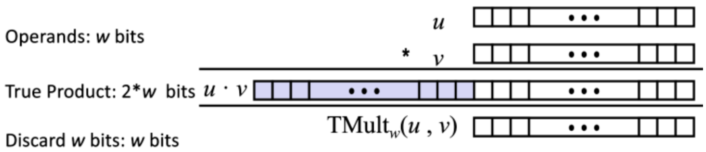
Unsigned Multiplication in C



- **Standard Multiplication Function**
  - Ignores high order w bits
- **Implements Modular Arithmetic**

  $\mathrm{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Signed Multiplication in C



- **Standard Multiplication Function**
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

- `u << k` gives `u * ` 2^k
- Both signed and unsigned

- `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
- Uses logical shift

在

没有现有的 1 损失的情况下

- **Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

$$-x = \sim x + 1$$

Learn to use masks in C programming

```
unsigned int x = 0x89ABCDEF;
unsigned int y = 0x76543210;
result = (x & 0x0000FFFF) | (y & 0xFFFF0000);
```

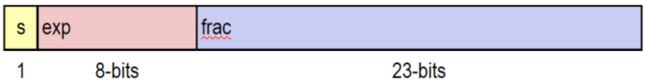
- Write a C expression that will yield a word consisting of the lowest two bytes of **x** and the highest two bytes of **y**. For operands **x** = 0x89ABCDEF and y = 0x76543210, this would give 0x7654CDEF.

0x89ABCDEF	0x76543210	0x0000CDEF
& 0x0000FFFF	& 0xFFFF0000	| 0x76540000
0x0000CDEF	0x76540000	0x7654CDEF

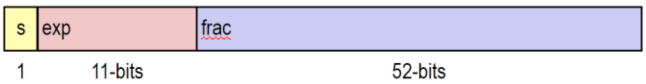
- Guess the output of the following C-language code. Note that short in C is 2-bytes on both 32-bit machine and 64-bit machine.
  - int x=100000;
  - short y = (short) x;
  - printf("%d\n",y);
- Explain your answer.

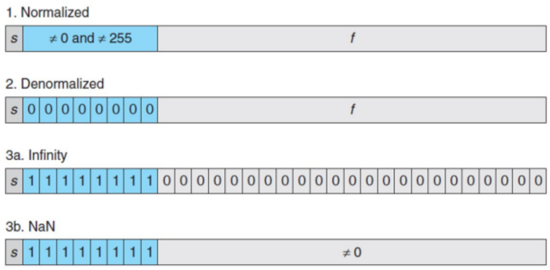
- x = 0x 0001 86A0, 4 bytes
- Casting a 4-byte int to a 2-byte short, the highest two bytes are lost by truncating, so y = 0x86A0
- 0x86A0 will be -31072, a signed integer

- **Single precision: 32 bits**



- **Double precision: 64 bits**





"Normalized" Values

$v = (-1)^s M\, 2^E$

- **When:** exp ≠ 000…0 and exp ≠ 111…1
- **Exponent** coded as a biased value: E = Exp – Bias
  - Exp: unsigned value of exp field
  - Bias = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
- **Significand** coded with implied leading 1: M = $1.xxx…x_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$v = (-1)^s M\, 2^E$
$E = Exp - Bias$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- **Significand**
  - M = $1.\underline{1101101101101}_2$
  - frac = $\underline{1101101101101}0000000000_2$
- **Exponent**
  - E = 13
  - $Bias$ = 127
  - Exp = $E + Bias$ = 140 = 10001100_2
- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

从数字转化为 normalized

Normalized Encoding Example

v = $(-1)^s M 2^E$
E = Exp – Bias

```
include <stdio.h>

ypedef unsigned char *pointer;

oid show_bytes(pointer start, size_t len){
 size_t i;
 for (i = 0; i < len; i++)
   printf("%p\t0x%.2x\n",start+i, start[i]);
 printf("\n");

nt main() {
   float a=15213;
   show_bytes((pointer)&a,sizeof(float));
   return 0;
```

Result on x86 (little Endian):

```
0x7ffcc3ce673c  0x00
0x7ffcc3ce673d  0xb4
0x7ffcc3ce673e  0x6d
0x7ffcc3ce673f  0x46
```

s	exp	frac
0	10001100	11011011011010000000000

从十六进制数据直接转化为 normalized，十六进制就是按 normalized 储存的

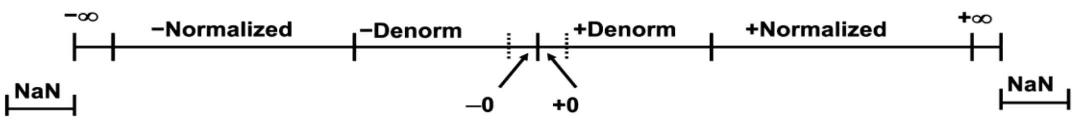
Denormalized Values

v = $(-1)^s M 2^E$
$E = 1 - Bias$

s	exp	frac

- **Condition**: exp = 000…0
- **Exponent** value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)
- **Significand** coded with implied leading 0: $M = 0.xxx…x_2$
  - **xxx…x**: bits of **frac**
- **Cases**
  - **exp** = 000…0, **frac** = 000…0
    - Represents zero value
  - **exp** = 000…0, **frac** ≠ 000…0
    - Numbers closest to 0.0

在 denormalized 中，E 和 Exp 没有关系，从 denormalized 转换为数字时，数字的 E=1-Bias，



Special Values

s	exp	frac

- **Condition: exp = 111…1**
- **Case: exp = 111…1, frac = 000…0**
  - Represents value ∞ (infinity)
  - Operation that **overflows**
  - E.g., the result of 1.0/0.0
- **Case: exp = 111…1, frac ≠ 000…0**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., the results of sqrt(–1), ∞ − ∞, ∞ × 0

Interesting Numbers

{single,double}

Description	*e xp*	*frac*	*Numeric Value*
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Single ≈ 1.4×10^{-45}			
Double ≈ 4.9×10^{-324}			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Single ≈ 1.18×10^{-38}			
Double ≈ 2.2×10^{-308}			
Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
Single ≈ 3.4×10^{38}	不可为 11…1 此为 Special Num		
Double ≈ 1.8×10^{308}			

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round up conditions**
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

- **Issue**
  - Rounding may have caused overflow
  - Handle by ~~shifting right once & incrementing exponent~~

Value	*Rounded*	*Exp*	*Adjusted*	*Result*
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

- **Default Rounding Mode**
  - Other rounding modes may produce statistically biased results
    - E.g., sum of positive numbers will consistently be over- or under-estimated
- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values
    - Round so that **least significant digit** is even
  - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

7.885，最近的偶数是 7.88；7.895，最近的偶数是 7.90

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly **overflow** if exponent too large
    - Possibly round to fit into **`frac`**

- $(-1)^{s1}\ M1\ 2^{E1} \times (-1)^{s2}\ M2\ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ 2^{E}$
  - Sign s: $s1$ ^ $s2$
  - Significand M: $M1 \times M2$
  - Exponent E: $E1 + E2$

s1	exp1	frac1
s2	exp2	frac2

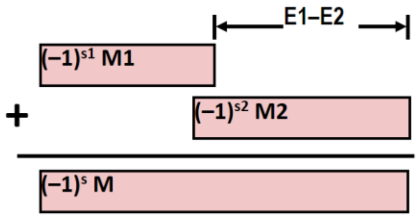
- **Fixing**
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, **overflow**
  - Round M to fit **`frac`** precision
- **Implementation**
  - The biggest task is multiplying significands

- **Multiplication Commutative?** *Yes*
- **Multiplication is Associative?** *No*
  - Possibility of overflow, inexactness of rounding
  - Ex: `(1e20*1e20)*1e-20= inf`,
    `1e20*(1e20*1e-20)=1e20`
- **1 is multiplicative identity?** *Yes*
- **Multiplication distributes over addition?** *No*
  - Possibility of overflow, inexactness of rounding
  - `1e20*(1e20-1e20)=0.0`,
    `1e20*1e20 – 1e20*1e20 =NaN`

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ +\ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume $E1 > E2$

Get binary points lined up



- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*
- **Fixing**
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Mathematical Properties of FP Add

- **Commutative?** Yes
- **Associative?** No
  - Overflow and inexactness of rounding
  - `(3.14+1e10)-1e10 = 0`,
    `3.14+(1e10-1e10) = 3.14`
- **0 is additive identity?** Yes
- **Every element has additive inverse?** Almost
  - Yes, except for infinities & NaNs

- A number that can be exactly represented in binary form will be written in the form $x/2^n$, where $x < 2^n$