Module1: Bits and Bytes, Integers, Floating Point

Bits, Bytes, Integers:

Bits: Made up by only 0 or 1.

Why bits: 1.Easy to store with bitstable elements

2.Reliably transmitted on noisy and inaccurate wires

For Example:

$15213_{10} = 11101101101101_2$

$1.20_{10} = 1.0011001100110011[0011]..._2$

$1.5213 \text{ X } 10^4 = 1.11011011011012 \text{ X } 2^{13}$

Binary: 0 or 1

Decimal: 0 to 9

Oct: 0 to 7

Hex: 0 to 9 and a,b,c,d,e,f

Fast way to transform between different encoding methods:

Decimal to Binary:

要点：除二取余，倒序排列

解释：将一个十进制数除以二，得到的商再除以二，依此类推直到商等于一或零时为止，倒取将除得的余数，即换算为二进制数的结果

例如把52换算成二进制数，计算结果如图：

```
2|  52   ................................0
  2|  26   ..............................0
    2|  13   ............................1
      2|  6   ...........................0
        2|  3   .........................1
             1   .........................1
```

52除以2得到的余数依次为：0、0、1、0、1、1，倒序排列，所以52对应的二进制数就是110100。

由于计算机内部表示数的字节单位都是定长的，以2的幂次展开，或者8位，或者16位，或者32位....。

于是，一个二进制数用计算机表示时，位数不足2的幂次时，高位上要补足若干个0。本文都以8位为例。那么：

(52)10=(00110100)2

Binary to Decimal:

整数二进制用数值乘以2的幂次依次相加，小数二进制用数值乘以2的负幂次然后依次相加！

比如将二进制110转换为十进制：

首先补齐位数，00000110，首位为0，则为正整数，那么将二进制中的三位数分别于下边对应的值相乘后相加得到的值为换算为十进制的结果

1	1	0
2^2	2^1	2^0

个位数 0 与 2^0 相乘：$0 \times 2^0 = 0$

十位数 1 与 2^1 相乘：$1 \times 2^1 = 2$

百位数 1 与 2^2 相乘：$1 \times 2^2 = 4$

将得到的结果相加：0+2+4=6

二进制 110 转换为十进制后的结果为 6

Decimal to other:

1. using the similar method as transforming to binary;

2. First, transform to binary, then use the below method:
   For example: 15213 = 0011 1011 0110 1101_2, transform to hex;
   $16 = 2^4$, for every four bytes:
   0011_2 = 0x3, 1011 = 0xb, 0110 = 0x6, 1101 = 0xd;
   So: 15213 = 0011 1011 0110 1101_2 = $0x3b6d_{16}$

Other to decimal: the same as binary to decimal.

Example Data Representations for Regular data type in C:

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
pointer	4	8	8

Boolean Algebra:

And(&): A & B only true when A = 1 and B = 1;

&	0	1
0	0	0
1	0	1

Or(|): A | B are true either when A = 1 or B = 1;

|	0	1
0	0	1
1	1	1

Exclusive-Or(XOR)(^): A ^ B only true when A = 1 or B = 1, but not both;

^	0	1
0	0	1
1	1	0

Not(~): ~A = 1 when A = 0;
~A = 0 when A = 1

Bit-Level Operations in C

Operations &, | , ~ , ^ are all available in C

1 Apply to any “integral” data type : **<u>long, int, short, char, unsigned</u>**

2 View arguments as bit vectors

3 Arguments applied bit-wise

Shift Operations:

1.**<u>Left Shift</u>**: x << y: Shift bit-vector x left y positions, throw away extra bits on left, and fill with 0's on right

2.**<u>Right Shift</u>**: x >> y: Shift bit-vector x right y positions, and throw away extra bits on right

Logical shift: Fill with 0's on left

Arithmetic shift: Replicate most significant bit on left

3. Undefined Behavior: Shift amount < 0 or $\geqslant$ word size

Encoding Integers With Sign-bit:

Unsigned:

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Signed:

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

The x_{w-1} is the sign bit;

For 2's complement, most significant bit indicates sign

1.0 for nonnegative

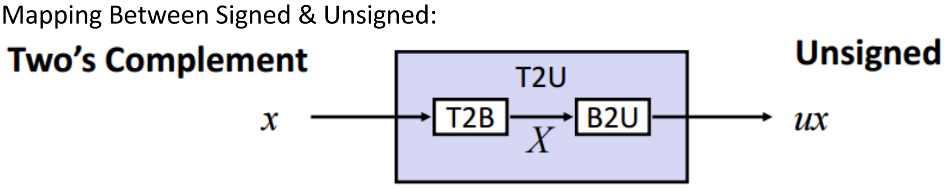
2.1 for negative

For Example: 10 = 01010_2, -10 = 10110_2, when the first bit is considered the sign-bit;

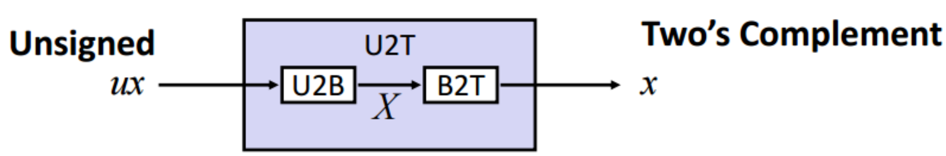
Method to calculate a negative integer A in binary: Find it's contract number A';

$A_2 = \sim A'_2 + 1$

Mapping Between Signed & Unsigned:



Maintain Same Bit Pattern

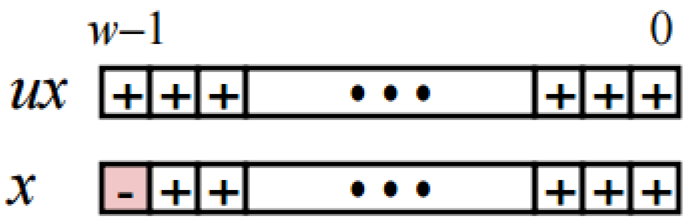


Maintain Same Bit Pattern

Mappings between unsigned and two's complement numbers:

Keep bit representations and reinterpret

Unsigned int doesn't have a sign-bit, but signed int has;



Expanding, truncating

Sign Extension
1.Task: Given w-bit signed integer x, Convert it to w+k-bit integer with same value
2.Rule: Make k copies of sign bit:
$X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x0$

Truncation
1.Task: Given k+w-bit signed or unsigned integer X, Convert it to w-bit integer X' with same value for "small enough" X
2.Rule: Drop top k bits:
$X; = x_{w-1}, x_{w-2}, \ldots, x0$

Summary:
Expanding, Truncating: Basic Rules
Expanding (e.g., short int to int)
1.Unsigned: zeros added
2.Signed: sign extension
3.Both yield expected result

Truncating (e.g., unsigned to unsigned short)
1.Unsigned/signed: bits are truncated
2.Result reinterpreted
3.Unsigned: mod operation
4.Signed: similar to mod
5.For small numbers yields expected behavior

Unsigned Addition
Standard Addition Function: Ignores carry output
Implements Modular Arithmetic:
$S = UAdd_w(u, v) = (u + v) \bmod 2^w$
Unsigned/signed: Normal addition followed by truncate, same operation on bit level
Unsigned: addition mod 2^w
Mathematical addition + possible subtraction of 2^w
Signed: modified addition mod 2w (result in proper range)
Mathematical addition + possible addition or subtraction of 2^w
Multiplication
Goal: Computing Product of w-bit numbers x, y (Either signed or unsigned)

1.Unsigned: up to 2wbits
(Result range: $0 \leqslant x * y \leqslant (2^w-1)^2 = 2^{2w} - 2^{w+1} + 1$)

2.Two’s complement min (negative): Up to 2w-1 bits
  Result range: $x * y \geqslant (-2^{w}-1) * (2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
3.Two’s complement max (positive): Up to 2wbits, but only for (TMinw)2
  Result range: $x * y \leqslant (-2^{w}-1)^{2} = 2^{2w-2}$

Unsigned/signed: Normal multiplication followed by truncate
Unsigned: multiplication mod 2^{w}
Signed: modified multiplication mod 2w (result in proper range)

<u>Byte-Oriented Memory Organization</u>
Programsrefer todata by address

Representing Strings
Represented by array of characters
1.Each character encoded in ASCII format
2.String should be null-terminated

<u>Floating Point</u>

<u>Fractional Binary Numbers</u>

Representation： $\sum_{k=-j}^{i} b_k \times 2^k$

Limitation:

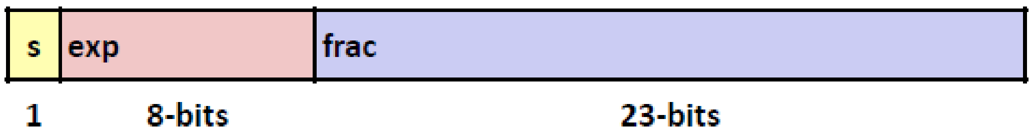
1. Can only exactly represent numbers of the form x/2k, Other rational numbers have repeating bit representations
2. Just one setting of binary point within the w bits, Limited range of numbers (very small values? very large?)

<u>IEEE Standard 754 to represent Floating Points:</u>
Numerical Form: **<u>$(-1)^{s} * M * 2^{E}$</u>**
<u>Sign bit(s)</u>: Determines whether number is negative or positive.
<u>Significand(M)</u>: Normally a fractional value in range [1.0,2.0).
<u>Exponent(E):</u> Weights value by power of two.

Encoding
1.MSB sis sign bit s
2.**exp** field encodes E(but is not equal to E)
3.**frac** field encodes M(but is not equal to M)

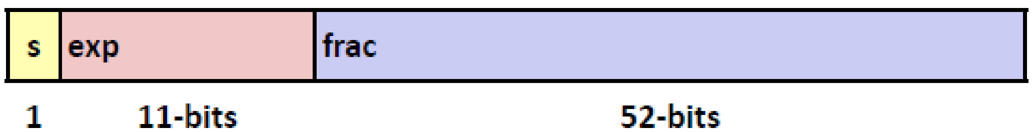
- Single precision: 32 bits

  ≈ 7 decimal digits, $10^{\pm 38}$



- Double precision: 64 bits

  ≈ 16 decimal digits, $10^{\pm 308}$



Usually, we use single precision.

Exponent coded asa biasedvalue: **E = Exp – Bias**

1.Exp: unsigned value of exp field

2.Bias = 2k-1-1, where kis number of exponent bits

Single precision: **127(Usually this in our homework or test)** (Exp: 1…254, E: -126…127)

Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

Significand coded with implied leading 1: M= 1.xxx…x2

1.xxx…x: bits of frac field

2.Minimum when frac = 000…0(M= 1.0)

3.Maximum when frac = 111…1(M= 2.0 –ε)

4.Get extra leading bit for "free"

Normalized Values: When: exp ≠ 000…0 and exp ≠ 111…1;

Denormalized Values: Condition: exp = 000…0, **for this, E = 1 – Bias, not 0 – Bias !!**

Special Values: 1. exp= 111…1, frac= 000…0: Represent **Infinity**, positive/negative depends on sign bit.

2. exp= 111⋯1, frac≠ 000⋯0: Represent **NaN**.

Special Properties ofthe IEEE Encoding

FP Zero Same as Integer Zero:

All bits = 0

Can (Almost) Use Unsigned Integer Comparison

1.Must first compare sign bits

2.Must consider −0 = 0

3.NaNs problematic

•Will be greater than any other values

•What should comparison yield? The answer is complicated.

4.Otherwise OK

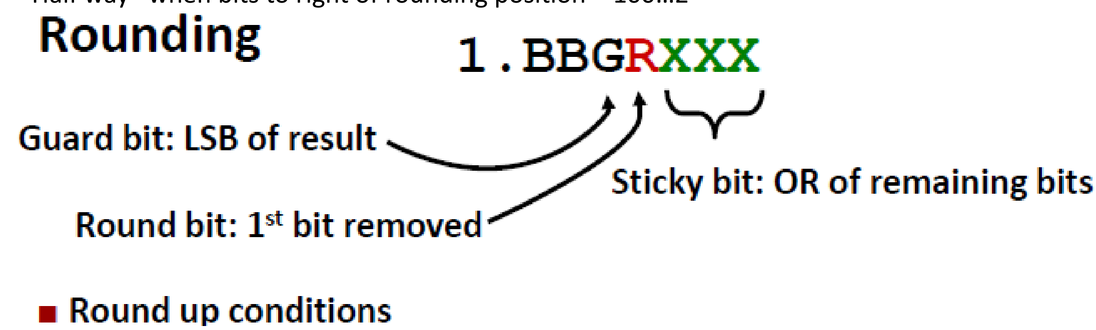
•Denormvs. normalized

•Normalized vs. infinity

Rounding Binary Numbers

Binary Fractional Numbers

- "Even" when least significant bit is 0
- "Half way" when bits to right of rounding position = $100\ldots_2$



Floating Points Multiplication

$(-1)^{s1} * M1 * 2^{E1} \times (-1)^{s2} * M2 * 2^{E2}$

- Exact Result: **$(-1)^s M 2^E$**
  - Sign s: s1^s2
  - Significand M: M1x M2
  - Exponent E: E1+E2
- Fixing
  - If $M \geqslant 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision
- Implementation: Biggest chore is multiplying significands

Floating Point Addition

- $(-1)^{s1} * M1 * 2^{E1} + (-1)^{s2} * M2 * 2^{E2}$: Assume E1> E2
- Exact Result: **$(-1)^s M 2^E$**
  - Sign s, significand M: Result of signed align & add
  - Exponent E: E1
- Fixing
  - If $M \geqslant 2$, shift M right, increment E
  - if M< 1, shift M left kpositions, decrement Eby k
  - Overflow if Eout of range
- Round M to fit frac precision