中国科学技术大学
University of Science and Technology of China

Computer Systems: A Programmer's Perspective
计算机系统

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

Part 1: Bits, Bytes, and Intergers

- **Representing information as bits**
- Bit-level manipulations
- Integers

Part 2：Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

从信号处理的角度：信息与数据

- **数据：未经解释或处理的原始信号或数值，是物理世界或数字世界的直接记录**
  - 无明确语义性 （如：传感器采集的电压/电流值、麦克风的声压采样点等）
- **信息（Information）的定义：**
  - 控制论创始人维纳（Norbert Wiener）认为“信息是人们在适应外部世界，并使这种适应反作用于外部世界的过程中，**同外部世界进行互相交换的内容**”
  - “内容”是事物的原形，” 交换“是信息载体将事物原形映射到人或其他物体的感觉器官，人们把这种映射的结果认为获得了”信息 “
  - 信息的具体表现形式称为”信号 “ （信息是信号包含的内容）
- **数据与信息的区别：**

维度	数据	信息
定义	原始信号的量化记录（物理/数字形式）	数据中隐含的语义或有意义的内容
核心属性	物理性、无明确意义、冗余/噪声	语义性、减少不确定性、依赖上下文
处理阶段	信号处理的输入	信号处理的输出
量度方式	用数量（如采样点、字节数）或存储衡量	用熵，互信息（Mutual Info）等度量
存在形式	具体的物理信号或离散数值（如电压像素值）	抽象的概念（如文本、图像内容等）

电子数字计算机

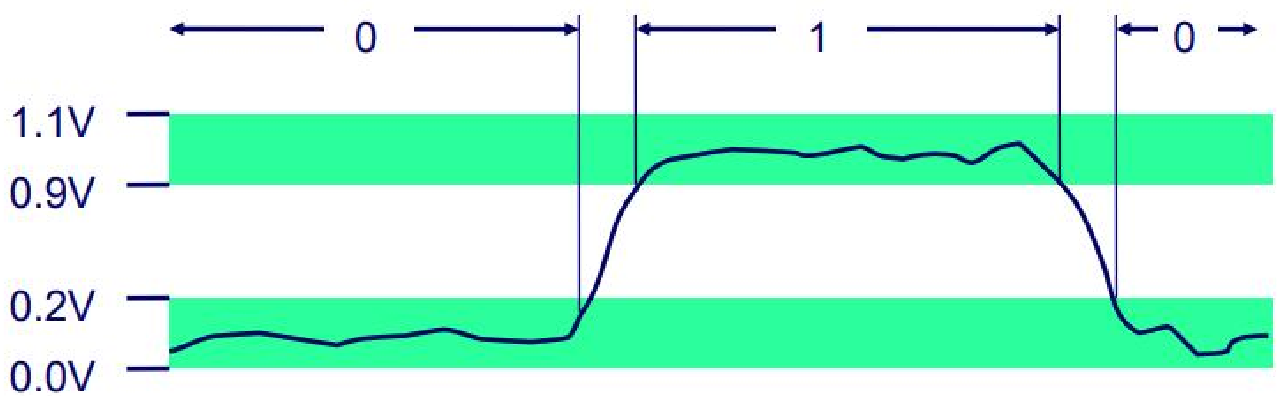
- **电子数字计算机：采用电子技术自动综合分析计算数值的精确的高速计算工具**
- **程序 = 算法 + 数据结构**
  - 算法：描述计算过程（指令序列）
  - “数据”结构：带有结构特性的“数据”元素的集合
- **程序的指令序列和“数据”：带有明确的语义性**

- **冯•诺伊曼(von Neumann)计算机（结构模型）**
  - 五大部件构成
  - 采用二进制表示指令和数据
  - 指令和数据以同等地位存放在存储器中，均可按地址访问 （信息的表示）
  - 运行过程：取指令、执行指令……
  - ....

计算机中如何表示程序设计语言中的信息?

Everything is bits

- **信息的表示：计算机中信息表示的基本单位为bit, 其取值为0或1，通过以不同方式编码/解释一组bits来表述不同的信息**
  – Computers determine what to do (instructions)
  – … and represent and manipulate numbers, sets, strings, etc…
- **为什么用 bits 表示? 主要从电子器件的实现角度考虑**
  – Easy to store with bistable elements **(易于存储)**
  – Reliably transmitted on noisy and inaccurate wires **(易于可靠传输)**



For example, can count in binary

- **以2为基数的数的表示**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Encoding Byte Values

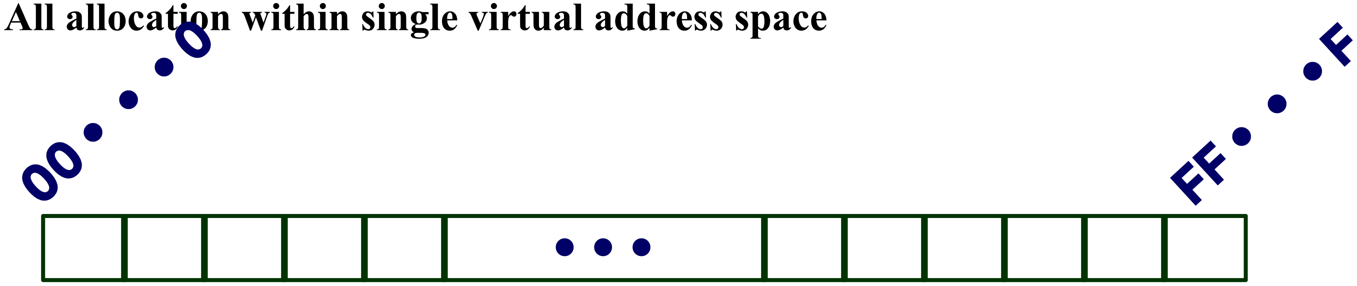
- **Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Octal: 000_8 to 0377_8
  - Hexadecimal: 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as `0xFA1D37B` or `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

15213: $\underbrace{0011}_{3}$ $\underbrace{1011}_{B}$ $\underbrace{0110}_{6}$ $\underbrace{1101}_{D}$

Byte-Oriented Memory Organization

- **程序涉及的内存地址通常是虚拟地址**
  - 虚拟地址空间：由进程的虚拟地址构成的空间
  - 概念上是非常大的（字节）数组 （模型）
  - 实际上是由不同存储介质构成的层次化的存储系统 （实体）
  - 系统为特定的“进程”提供私有存储（地址）空间
    - 程序在其私有空间中运行，可以修改其私有空间的数据，但是不能直接修改其他（地址空间）的数据
- **内存空间由编译器+运行时系统来控制分配**
  - **Where different program objects should be stored**
  - **All allocation within single virtual address space**

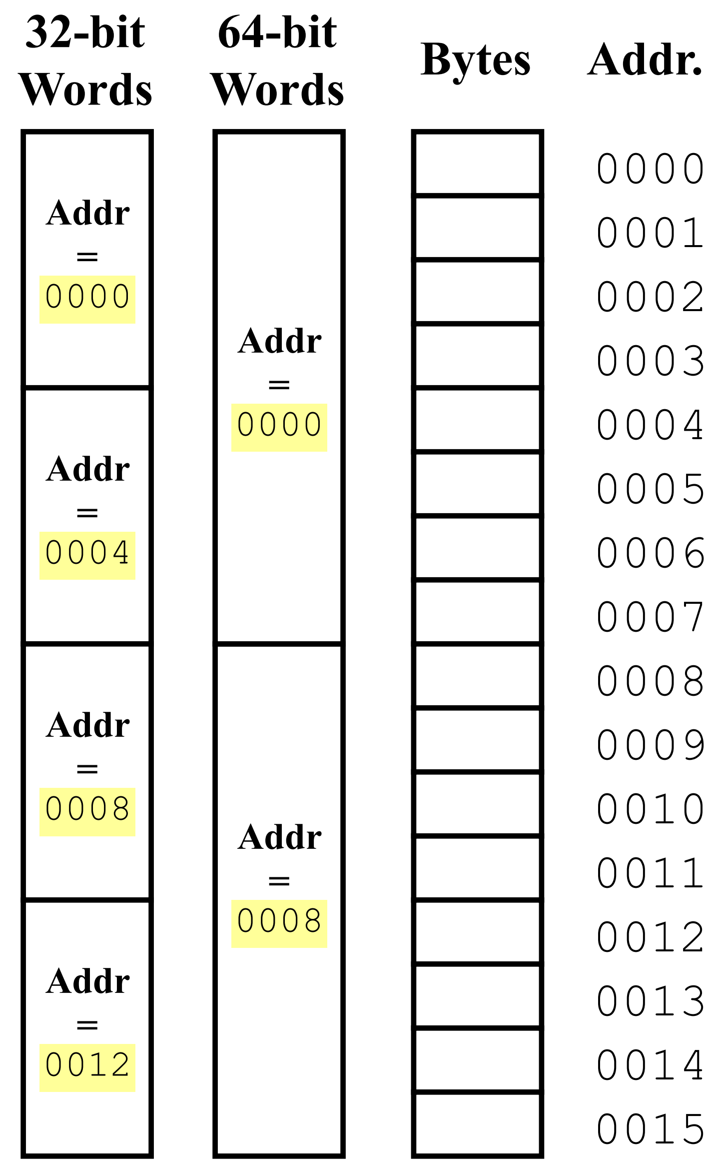


Machine Words

- **计算机系统中“字长”的概念(Word Size)**
  - 通常用整型数表示（位数、字节数）
    - 机器字长、指令字长、数据字长、地址字长
    - csapp：高级语言（C）中指针数据标称的位数
      - 反映程序可访问的虚拟存储空间的大小
  - 有些机器字长32bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - 大多数通用计算机使用64 bits (8 bytes) 字长
    - Potential address space ≈ 1.8×10^{19} bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
  - 许多机器支持多种数据格式（不同字节数)
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **存储器按字节编址，字地址描述了字中某一字节的位置**
  - 字中首字节地址
  - 连续字地址相差4（32位）或8（64位）
- **高级语言中不同数据类型所占的字节数不同**



Example Data Representations

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

Byte Ordering

- **多字节“字”中的字节在内存中应该如何排列?**
- **两种约定**
  - **Big Endian: Sun, PPC , Mac, Internet**
    - **Most significant byte has lowest address**
  - **Little Endian: x86, ARM processors running Android, iOS, and Linux**
    - **Least significant byte has lowest address**
  - **Bi-Endian: Big Endian or Little Endian**

Byte Ordering Example

- **Big Endian （大尾端/大端）**
  - Most significant byte has lowest address
- **Little Endian（小尾端/小端）**
  - Least significant byte has lowest address
- **Example**
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

高位

Big Endian：从高位字节开始存储

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

高位低地址字节

Little Endian：从低位字节开始存储

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

低位低地址字节

Reading Byte-Reversed Listings

- **反汇编**
  - 二进制机器代码的文本表示
  - 一般由读取机器代码的程序生成
- **反汇编码的例子**

Address	Instruction Code	Assembly Rendition
`8048365:`	`5b`	`pop %ebx`
`8048366:`	`81 c3 ab 12 00 00`	`add $0x12ab,%ebx`
`804836c:`	`83 bb 28 00 00 00 00`	`cmpl $0x0,0x28(%ebx)`

Deciphering Numbers

- **Value:** `0x12ab`
- **Pad to 4 bytes:** `0x000012ab`
- **Split into bytes:** `00 00 12 ab`
- **Reverse:** `ab 12 00 00`

review

- **信息（Information）的定义：**
  - 信息是人们同外部世界进行互相交换的内容
  - 内容是事物的原形，交换是信息载体将事物原形映射到人或其他物体的感觉器官，人们把这种映射的结果认为获得了信息
  - 信息的具体表现形式称为信号（信息是信号包含的内容）
- **计算机中如何表示程序设计语言中的信息?**
  - Everything is bits
  - 基本单位为bit, 其取值为0或1
  - 通过以不同方式编码/解释一组bits来表述不同的信息
    - 指令、数据、地址
    - 程序编码为指令序列
  - 内存以字节为单位编址
  - 计算机中“字”的地址：Big Endian（**从字的高位字节开始存储**）、Little Endian（**从字的低位字节开始存储**）

Examining Data Representations

- 打印“某数据”字节表示的代码
  - Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char *pointer;

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

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

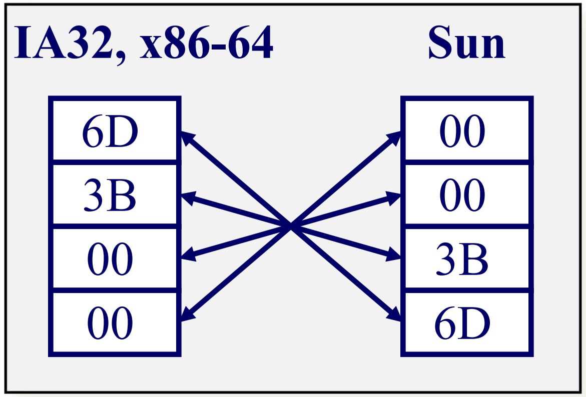
Result (Linux): Little Endian

```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```

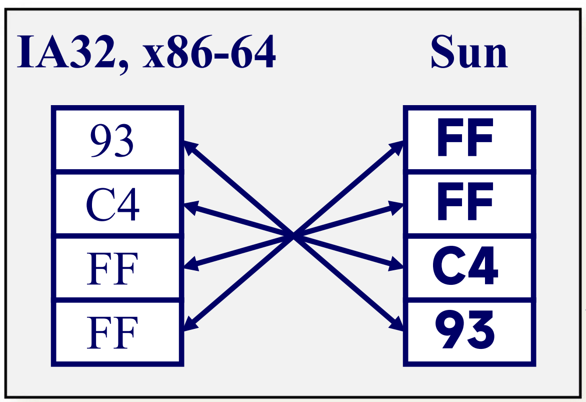
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

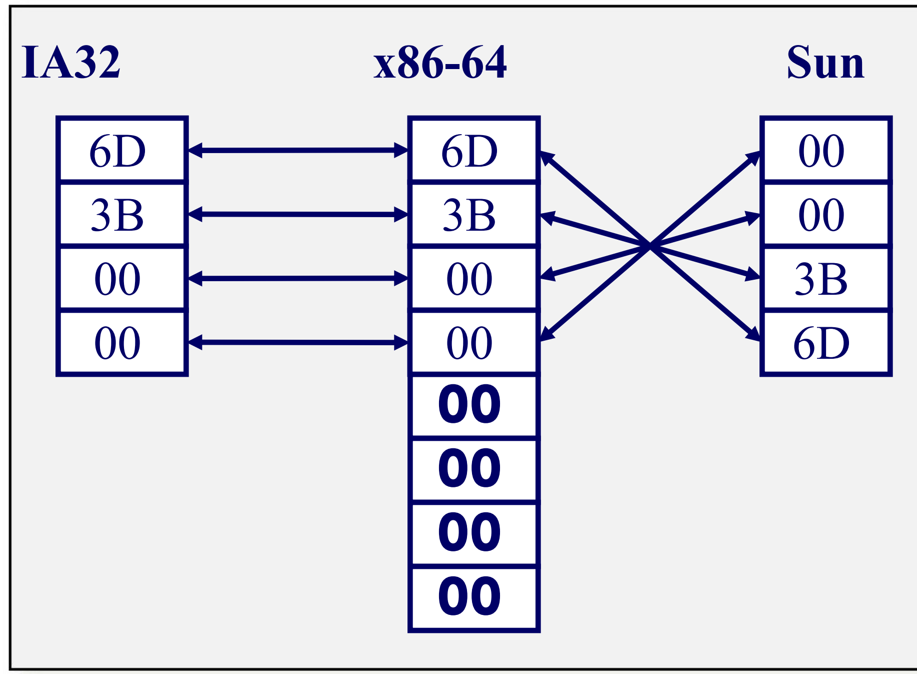
int A = 15213;



int B = -15213;



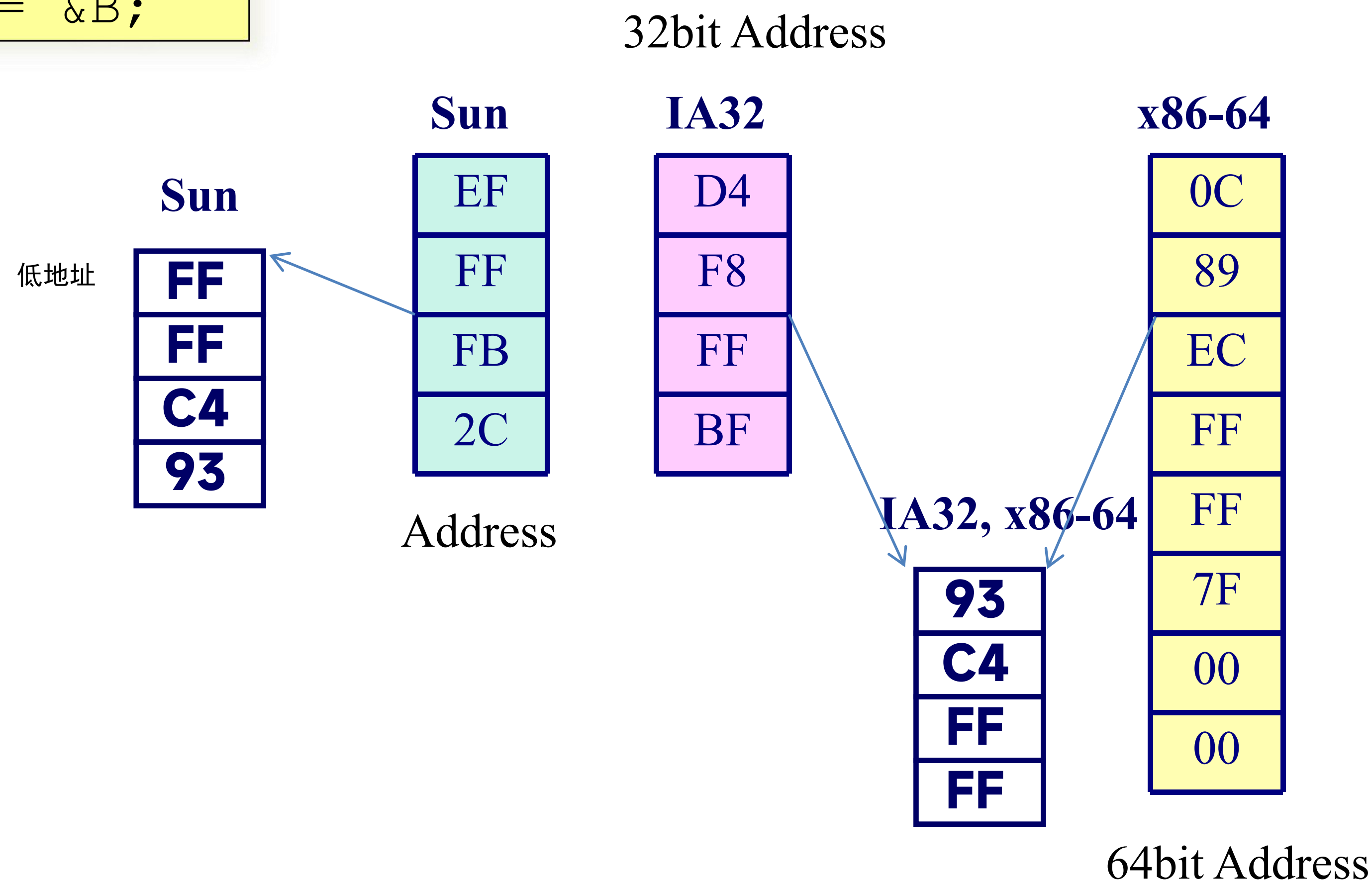
long int C = 15213;



Two's complement representation
(Covered later)

Representing Pointers

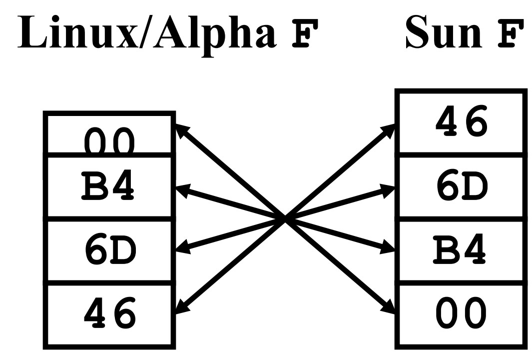
```
int B = -15213;
int *P = &B;
```



Different compilers & machines assign different locations to objects

Representing Floats

- `Float F = 15213.0;`



IEEE Single Precision Floating Point Representation

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
15213:			1110	1101	1011	01		

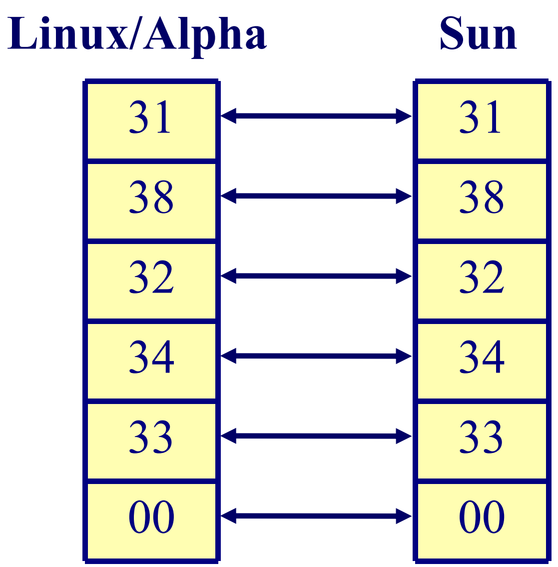
Not same as integer representation, but consistent across machines

Can see some relation to integer representation, but not obvious

Representing Strings

```
char S[6] = "18243";
```

- **C语言中 字符串的表示**
  - **Represented by array of characters**
  - **Each character encoded in ASCII format**
    - **Standard 7-bit encoding of character set**
    - **Character "0" has code 0x30**
      - Digit i has code 0x30+i
    - **String should be null-terminated**
      - **Final character has code 0x00**
- **字符串表示的兼容性问题?**
  - Byte ordering not an issue



Machine-Level Code Representation

- **将程序编码为指令序列**
  - Each simple operation
    - Arithmetic operation
    - Read or write memory
    - Conditional branch
  - Instructions encoded as bytes
    - Alpha’s, Sun’s, Mac’s use 4 byte instructions
      - Reduced Instruction Set Computer (RISC)
    - PC’s use variable length instructions
      - Complex Instruction Set Computer (CISC)
  - Different instruction types and encodings for different machines
    - Most code not binary compatible
- **程序也是字节序列!**

Representing Instructions

```
int sum(int x, int y)
{
    return x+y;
}
```

- 本例中Alpha & Sun 使用2条4字节指令
  - Use differing numbers of instructions in other cases
- PC 使用7条长度不等的指令（1，2，3字节指令）
  - Same for NT and for Linux
  - NT / Linux not fully binary compatible

Alpha sum	Sun sum	PC sum
00	81	55
00	C3	89
30	E0	E5
42	08	8B
01	90	45
80	02	0C
FA	00	03
6B	09	45
		08
		89
		EC
		5D
		C3

Different machines use totally different instructions and encodings

Review: Representing information as bits

- **信息表示的基本单位为bit, 其取值为0或1**
- **通过以不同方式编码/解释一组bits来表示不同的信息**
  - 指令、数据、地址
  - 程序编码为指令序列
- **内存以字节为单位编址**
- **计算机中“字”的地址**
  - Big Endian、Little Endian

第2章　信息的表示和处理
Part 1: Bits, Bytes, and Intergers

- Representing information as bits
- **Bit-level manipulations**
- Integers
- Representations in memory, pointers, strings

Boolean Algebra

- **19世纪George Boole提出了布尔代数**
  - 逻辑的代数表示：Encode “True” as 1 and “False” as 0
  - 用数学方法来刻画和验证逻辑推理的规律

- **And**
- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

- **Or**
- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

- **Not**
- **~A = 1 when A=0**

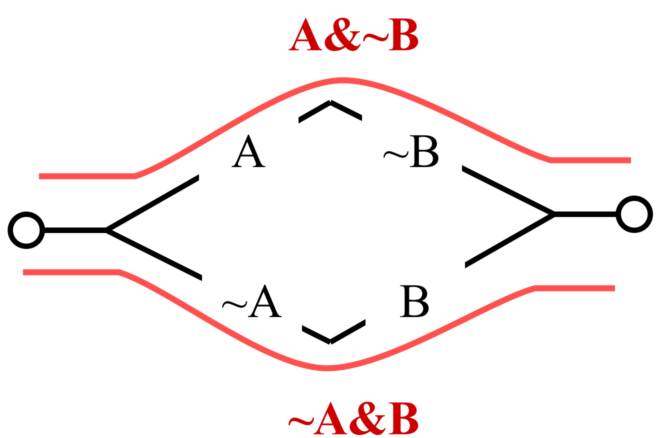
~	
0	1
1	0

- **Exclusive-Or (Xor)**
- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

Application of Boolean Algebra

- **Claude Shannon将布尔代数应用于数字系统**
  - 1937 MIT Master’s Thesis
  - 用于 关于继电器开关网络的推理
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

General Boolean Algebras

- **位向量（Bit Vectors） 操作（按位诸位操作）**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **满足布尔代数的所有性质**
  - **A&B = B&A, A|B = B|A （交换律）**
  - **A&(B|C) = (A&B) | (A&C), A | (B&C) = (A|B) & (A|C) （分配律）**
  - **A | 0 = A, A&1 = A （单位元）**
  - **A|~A = 1, A&~A = 0 （互补律）**
  - **幂等律、德.摩根律、零一律、对合律、囿元律.....**

Representing & Manipulating Sets

- **位向量可以表示（有约束的）有限集合**
  - Width w bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ if $j \in A$

    `01101001` { 0, 3, 5, 6 }

    `76543210`

    `01010101` { 0, 2, 4, 6 }

    `76543210`

- **位向量操作与集合操作**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **C语言中可用的位操作 &, |, ~, ^**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **例如 (Char data type)**
  - `~0x41 --> 0xBE`

    $\sim 01000001_2 \; \texttt{-->} \; 10111110_2$
  - `~0x00 --> 0xFF`

    $\sim 00000000_2 \; \texttt{-->} \; 11111111_2$
  - `0x69 & 0x55 --> 0x41`

    $01101001_2 \,\&\, 01010101_2 \; \texttt{-->} \; 01000001_2$
  - `0x69 | 0x55 --> 0x7D`

    $01101001_2 \mid 01010101_2 \; \texttt{-->} \; 01111101_2$

Contrast: Logic Operations in C

- **C语言中位运算符与逻辑运算符的差异**
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **例如 (char data type)**
  - `!0x41  -->  0x00`
  - `!0x00  -->  0x01`
  - `!!0x41 -->  0x01`

  - `0x69 && 0x55  -->  0x01`
  - `0x69 || 0x55  -->  0x01`
  - `p && *p  (avoids null pointer access)`

Shift Operations

- **左移操作: x << y**
  – Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **右移操作: x >> y**
  – Shift bit-vector x right y positions
    - Throw away extra bits on right
  – Logical shift
    - Fill with 0's on left
  – Arithmetic shift
    - Replicate most significant bit on right
    - Useful with two's complement integer representation
- **未定义的行为（Undefined Behavior)**
  – Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Cool Stuff with Xor

- **按位Xor可视为一种加法方式**
- **其加性逆元是它本身**

$$A \wedge A = 0$$

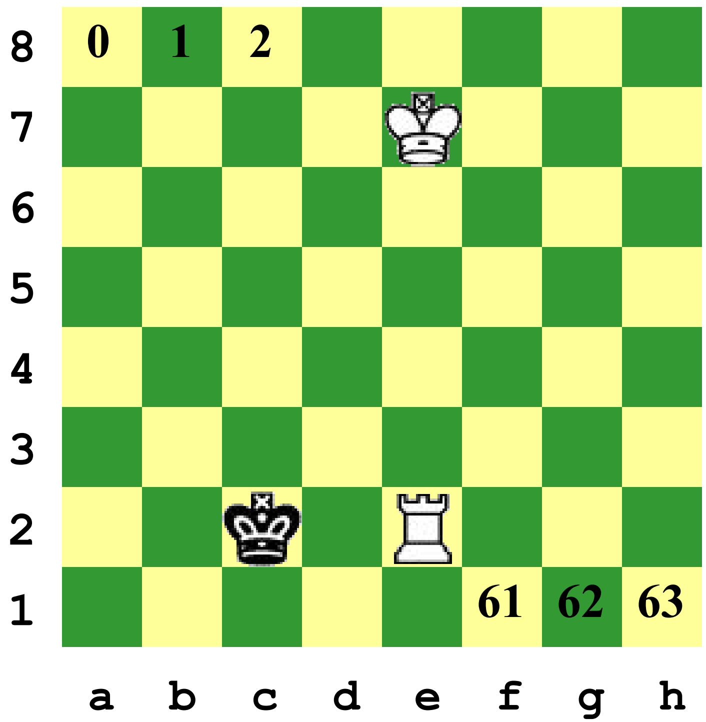
```
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A
3	(A^B)^A = B	A
End	B	A

More Fun with Bitvectors

- **国际象棋的Bit-board表示:**

```
unsigned long long blk_king, wht_king,
  wht_rook_mv2,…;
```



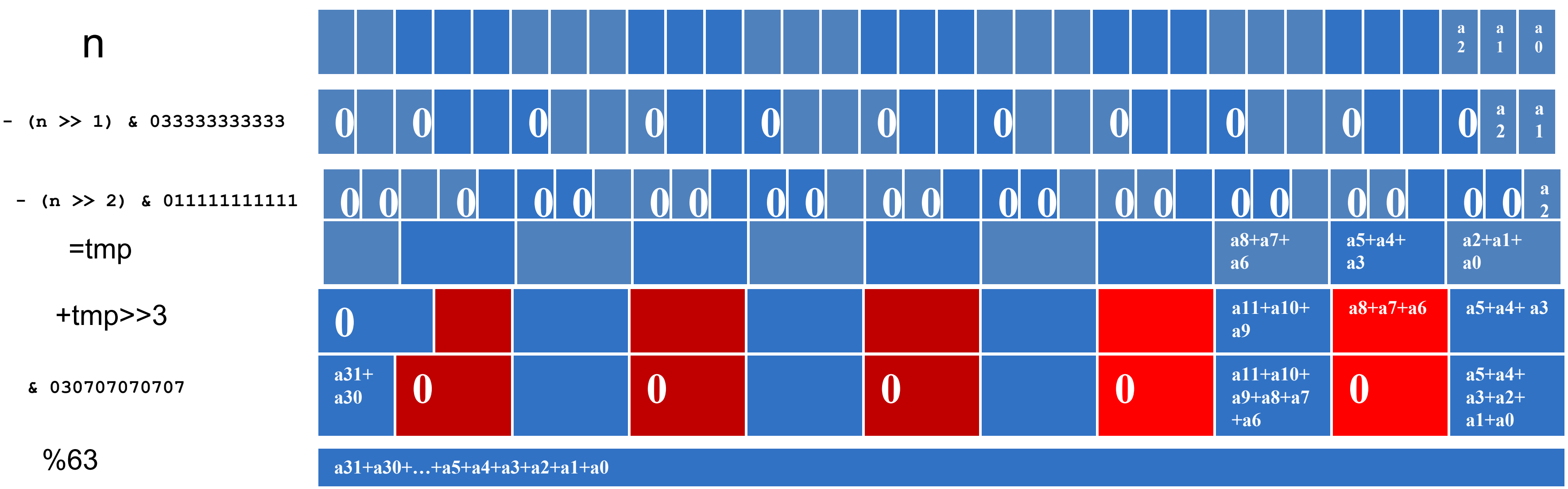

```
wht_king      = 0x0000000000001000ull;
blk_king      = 0x0004000000000000ull;
wht_rook_mv2 = 0x10ef101010101010ull;
...
/*
 * Is black king under attach from
 * white rook ?
 */
if (blk_king & wht_rook_mv2)
    printf(”Yes\n”);
```

More Bitvector Magic

- **计算一个word中1的个数**MIT Hackmem 169:

```
int bitcount(unsigned int n)
{
    unsigned int tmp;

    tmp = n - ((n >> 1) & 033333333333)
            - ((n >> 2) & 011111111111);
    return ((tmp + (tmp >> 3)) & 030707070707)%63;
}
```

Tips: 常数10
二进制表示:
0b1010
八进制表示
012
16进制表示:
0x10



Some Other Uses for Bitvectors

- **位向量可表示元素较少的集合**
- **可用来表示多项式**
  - Important for error correcting codes
  - Arithmetic over finite fields, say GF(2^n)
  - Example 0x15213 : $x^{16} + x^{14} + x^{12} + x^9 + x^4 + x + 1$
- **图的表示**
  - A ‘1’ represents the presence of an edge
- **位图图像、图标、光标的表示…**
  - Exclusive-or cursor patent
- **布尔表达式和逻辑电路的表示**

Summary of the Main Points

- **数、程序、文本用 Bits & Bytes表示**
- **不同机器对字长、尾端以及信息表示的方式有不同的约定**
- **数学基础是布尔代数**
  - 基本形式将false编码为0，true编码为1
  - 一般形式：例如C语言中的位级操作
    - 适合 表示和操作 集合

- Representing information as bits
- Bit-level manipulations
- **Integers**

Integers outline

- **Representation: unsigned and signed**
- **Conversion, casting**
- **Expanding, truncating**
- **Addition, negation, multiplication, shifting**
- **Summary**

C Puzzles

- **假设机器字长：32位；采用2的补码表示整型数**
- **试回答下列C语言表达式是否对任意值均为真，如果不为真，请举例说明**

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

Terminology for integer data

Symbol	Type	Meaning
B2T	Function	Binary to two's complement
B2U	Function	Binary to unsigned
U2B	Function	Unsigned to binary
U2T	Function	Unsigned to two's complement
T2B	Function	Two's complement to binary
T2U	Function	Two's complement to unsigned
TMin	Constant	Minimum two's complement value
TMax	Constant	Maximum two's complement value
UMax	Constant	Maximum unsigned value

B: 二进制位模式 T: 补码表示 U：无符号数表示

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

- **C语言中short int型占2个字节**

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

- **C 语言规范中没有强制要求采用2的补码表示**
  - **But, most machines do, and we will assume so**
- **符号位**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative， 1 for negative

Two-complement: Simple Example

Two's Complement

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

Sign Bit

		-16	8	4	2	1	
10	=	0	1	0	1	0	8+2 = 10

		-16	8	4	2	1	
-10	=	1	0	1	1	0	-16+4+2 = -10

Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

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

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

- **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 bits

	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

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **观察到的现象：**
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
  - **Question: abs(TMin)?**

C Programming

```
#include <limits.h>
```

K&R App. B11

Declares constants, e.g.,

```
ULONG_MAX
LONG_MAX
LONG_MIN
```

Values platform-specific

Unsigned & Signed Numeric Values

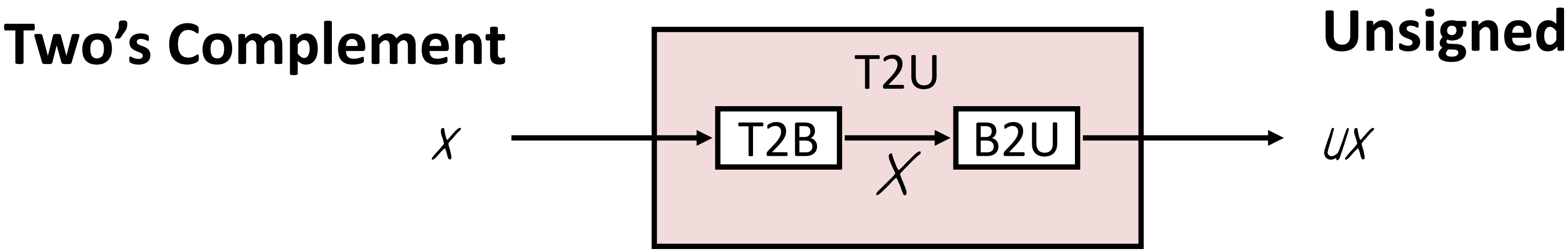
X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **等价性(Equivalence)**
  - Same encodings for nonnegative values
- **唯一性(Uniqueness)**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **⇒ 逆映射(Invert Mapping)**
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer

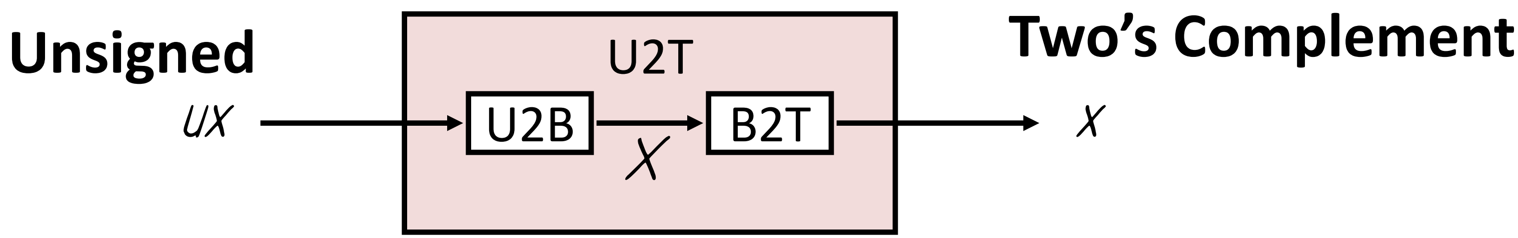
Integers outline

- Representation: unsigned and signed
- **Conversion, casting**
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Mapping Between Signed & Unsigned
底层映射规则



Maintain Same Bit Pattern X



Maintain Same Bit Pattern X

- **无符号整数与2的补码（有符号整数）之间的映射:**
  keep bit representations and reinterpret

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

$\longrightarrow \boxed{\text{T2U}} \longrightarrow$

$\longleftarrow \boxed{\text{U2T}} \longleftarrow$

Mapping Signed ↔ Unsigned

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

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

Bits	Signed		Unsigned
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	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4	⟷ (+/- 2^w)	12
1101	-3		13
1110	-2		14
1111	-1		15

Casting Signed to Unsigned

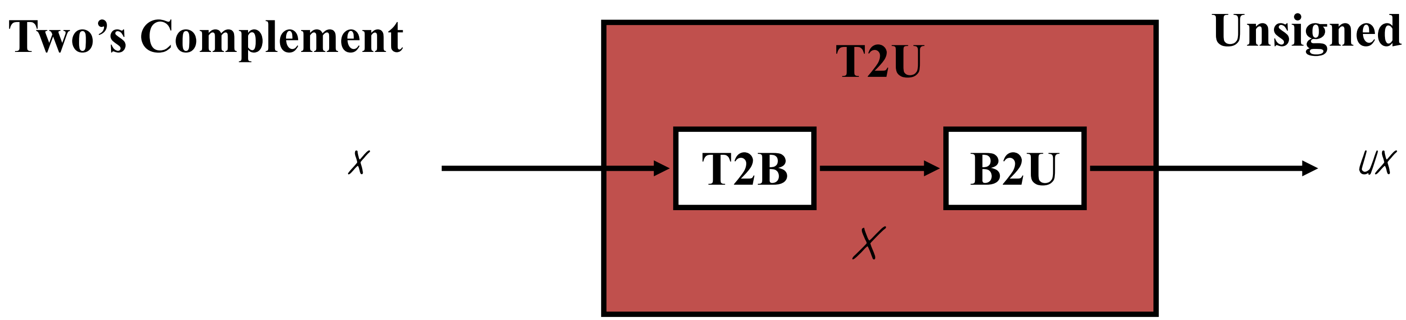
- **C语言允许有符号整型数转换为无符号整型数**

```
short int            x =  15213;
unsigned short int ux = (unsigned short) x;
short int            y = -15213;
unsigned short int uy = (unsigned short) y;
```

- **转换结果**
  - No change in bit representation
  - Nonnegative values unchanged
    - $ux = 15213$
  - **Negative values change into (large) positive values**
    - $uy = 50323$
      $= \underline{2^{16} + y}$

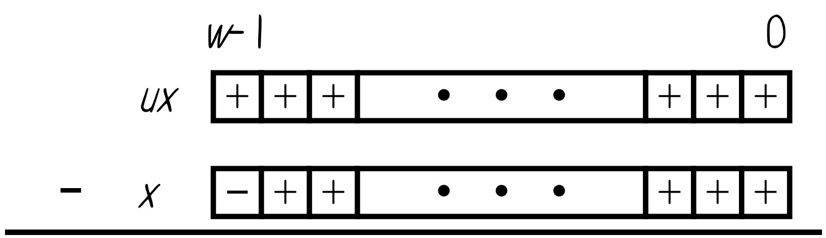
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

Relation between Signed & Unsigned



Maintain Same Bit Pattern

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



$$+2^{w-1} - -2^{w-1} = 2*2^{w-1} = 2^w$$

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

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

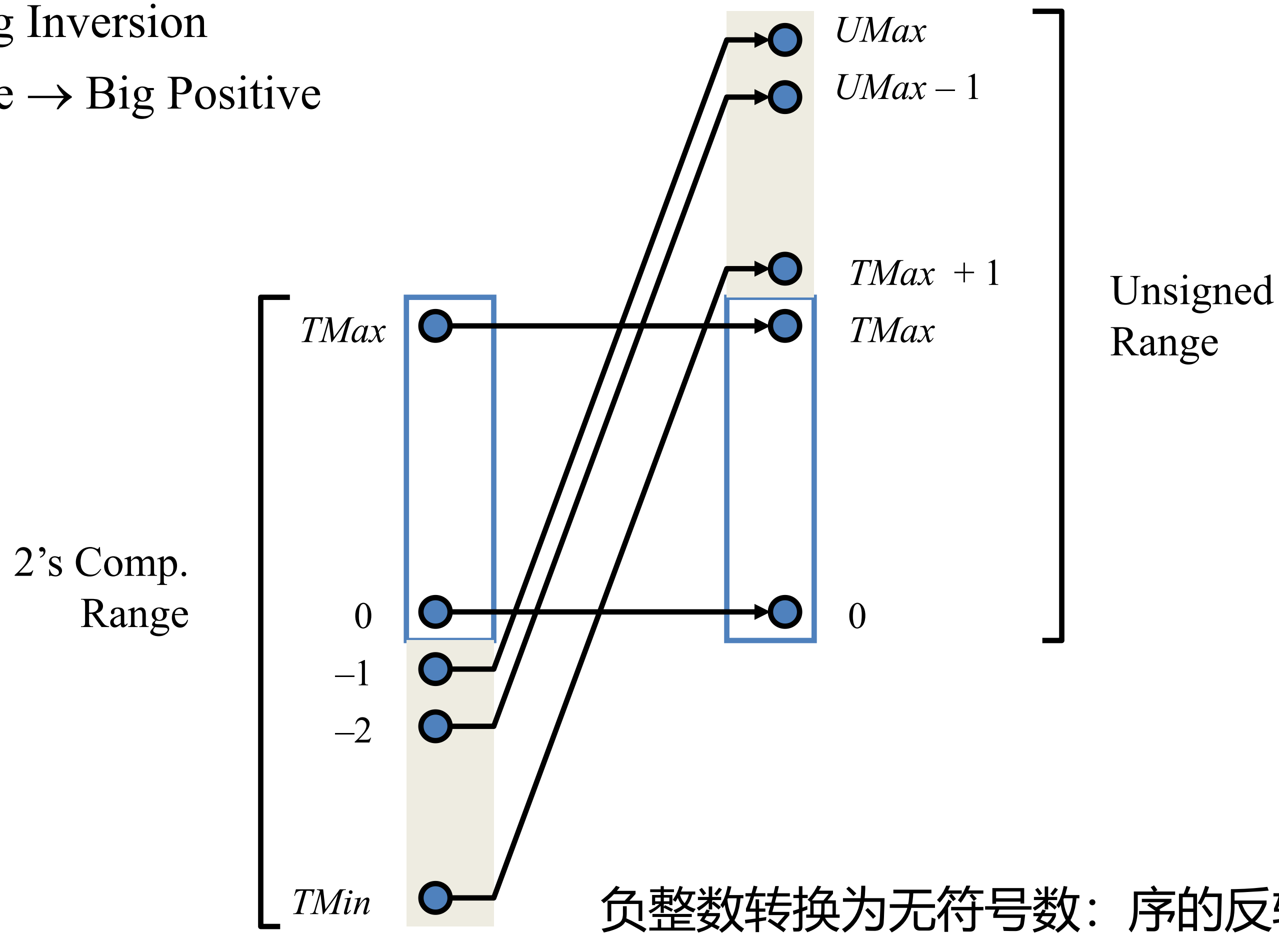
$$ux = \begin{cases} x & x \ge 0 \\ x + 2^w & x < 0 \end{cases}$$

Relation Between Signed & Unsigned

Weight	-15213		50323	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
32768	1	*-32768*	1	*32768*
Sum		**-15213**		**50323**

Explanation of Casting Surprises

- **2的补码→ 无符号数**
  - Ordering Inversion
  - Negative → Big Positive



负整数转换为无符号数：序的反转

Signed vs. Unsigned in C

- **C 语言中的常量(Constants)**
  - **By default are considered to be signed integers**
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- **类型转换(Casting)**
  - Explicit casting between signed & unsigned same as U2T and T2U

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

  - Implicit casting also occurs via assignments and procedure calls

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

Casting Surprises

- **表达式求值中类型转换规则**
  – 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 = -2,147,483,648 , TMAX = 2,147,483,647*

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Summary
Casting Signed ↔ Unsigned: Basic Rules

- **相互转换时保持位模式不变，但按新的数据类型表示方式解释**

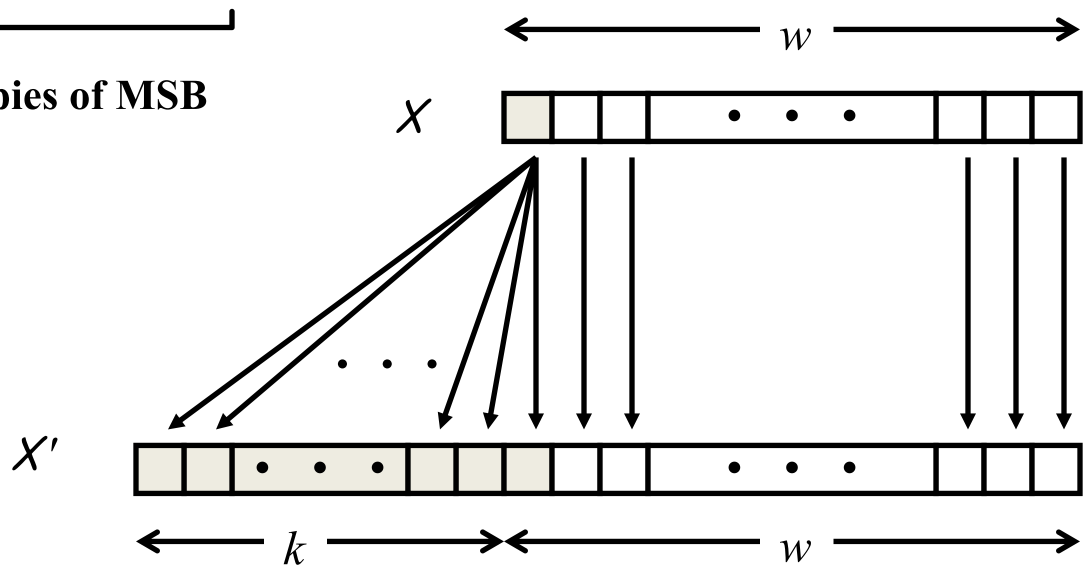
- **可能会有意想不到的效果: 加或减 2^w**
  - 位模式最高位为1时
- **包含有符号和无符号整型的表达式时：**
  - **int is cast to unsigned!!**

Integers outline

- Representation: unsigned and signed
- Conversion, casting
- **Expanding, truncating**
- Addition, negation, multiplication, shifting
- Summary

Sign Extension

- **有符号整型数位扩展:**
  - Given w-bit signed integer x
  - Convert it to w+k-bit integer **with same value**
- **位扩展规则:**
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension: Simple Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Sign Extension Example

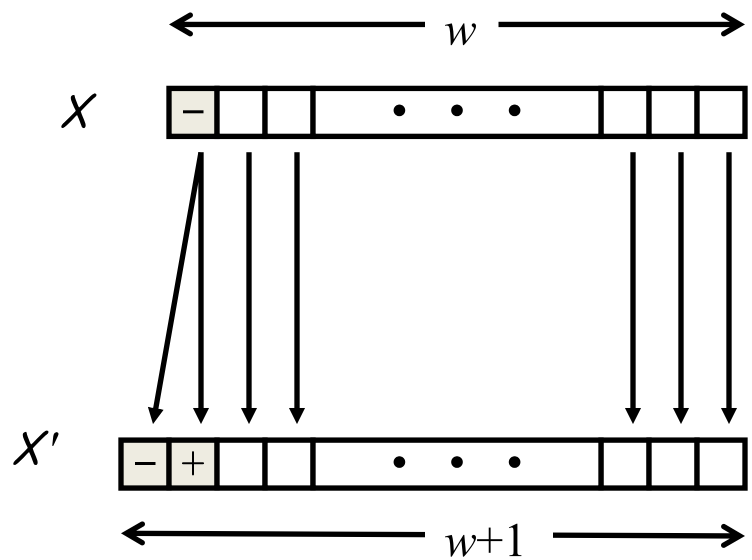
```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- **从较小的到较大的(有符号)整型数据类型的转换**
- **C 自动执行符号位扩展**

Justification For Sign Extension

- **用归纳法证明符号位扩展的正确性**
  - Induction Step: extending by single bit maintains value



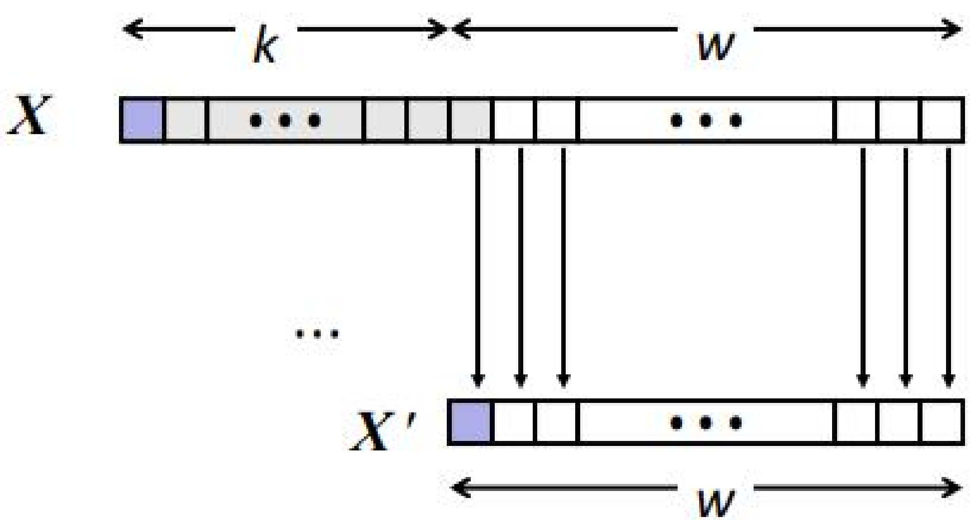
  - Key observation: $-2^{w-1} = -2^{w} + 2^{w-1}$
  - Look at weight of upper bits:

$X \qquad -2^{w-1}\, x_{w-1}$

$X' \qquad -2^{w}\, x_{w-1} + 2^{w-1}\, x_{w-1} \quad = \quad -2^{w-1}\, x_{w-1}$

Truncation

- **整型数的截断（Truncation）操作:**
  - Given k+w-bit signed or unsigned integer X
  - Convert it to w-bit integer X’ with same value for **“small enough”** X
- **截断操作规则:**
  - Drop top k bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation: Simple Example

No sign change

2 =

-16	8	4	2	1
0	0	0	1	0

2 =

-8	4	2	1
0	0	1	0

2 mod 16 = 2

-6 =

-16	8	4	2	1
1	1	0	1	0

-6 =

-8	4	2	1
1	0	1	0

-6 mod 16 = 26U mod 16 = 10U = -6

Sign change

10 =

-16	8	4	2	1
0	1	0	1	0

-6 =

-8	4	2	1
1	0	1	0

10 mod 16 = 10U mod 16 = 10U = -6

-10 =

-16	8	4	2	1
1	0	1	1	0

6 =

-8	4	2	1
0	1	1	0

-10 mod 16 = 22U mod 16 = 6U = 6

Summary:
Expanding, Truncating: Basic Rules

- **整型数位扩展 (Expanding: e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- **整型数位截断(Truncating: e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Integers outline

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- **Addition, negation, multiplication, shifting**
- Summary

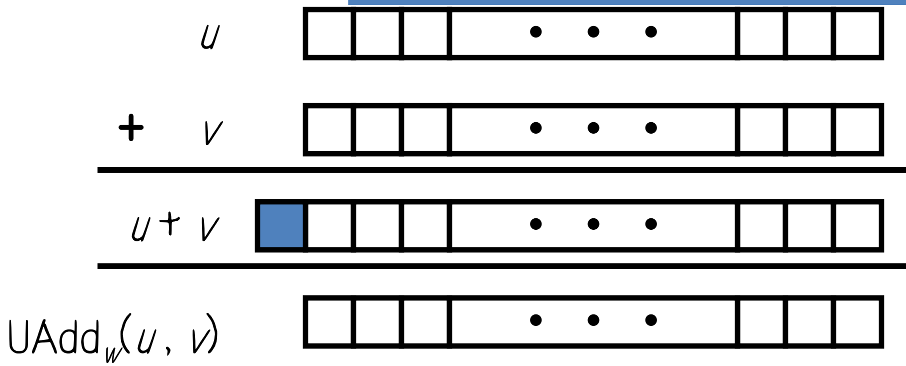
Unsigned Addition

$$UAdd_w(u,v) = \begin{cases} u+v & u+v<2^w \\ u+v-2^w & u+v\geq 2^w \end{cases}$$

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- **标准的忽略进位的加法运算**
  - Ignores carry output
- **实现的是模运算**

$$s \;=\; \mathrm{UAdd}_w(u\ ,\ v) \;=\; u + v \mod 2^w$$

CF ZF SF OF

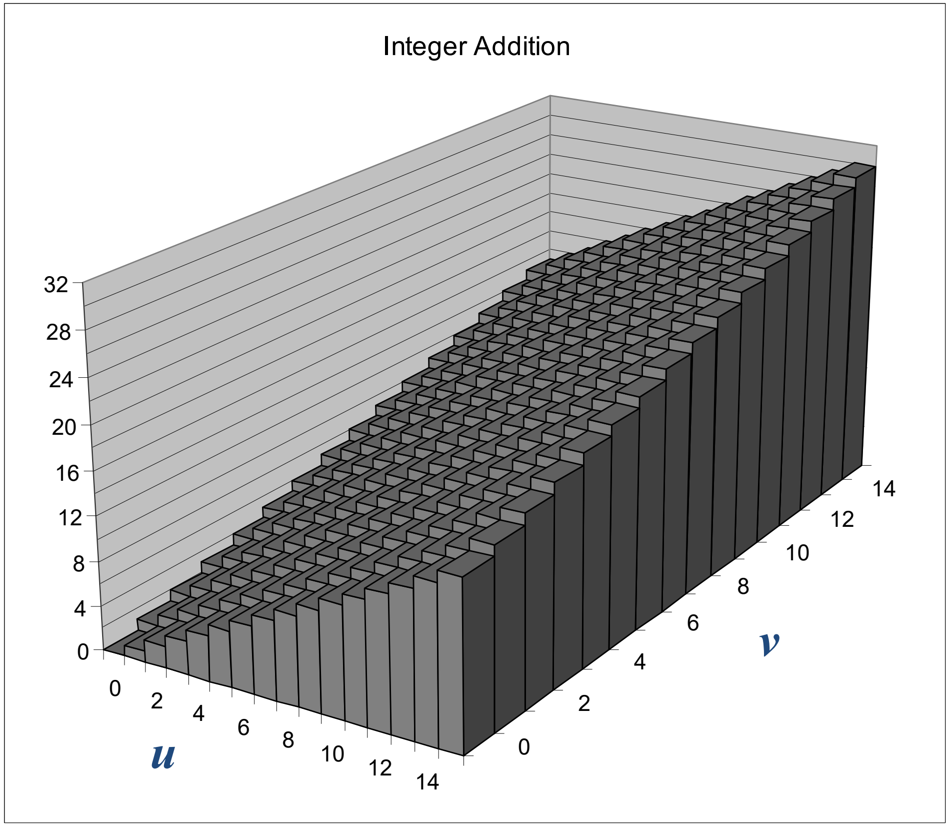
Condition codes / Status

unsigned char	1110 1001	E9	223
	+ 1101 0101	+ D5	+ 213
	1 1011 1110	1BE	446
	1011 1110	BE	190

Visualizing (Mathematical) Integer Addition

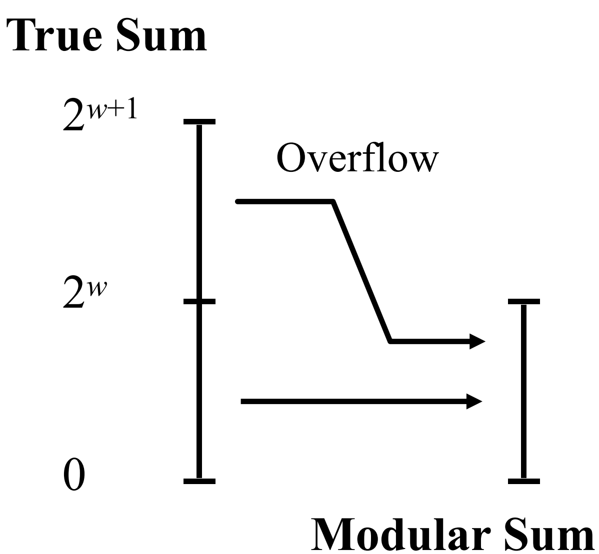
- **非负整数相加**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u\ ,\ v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

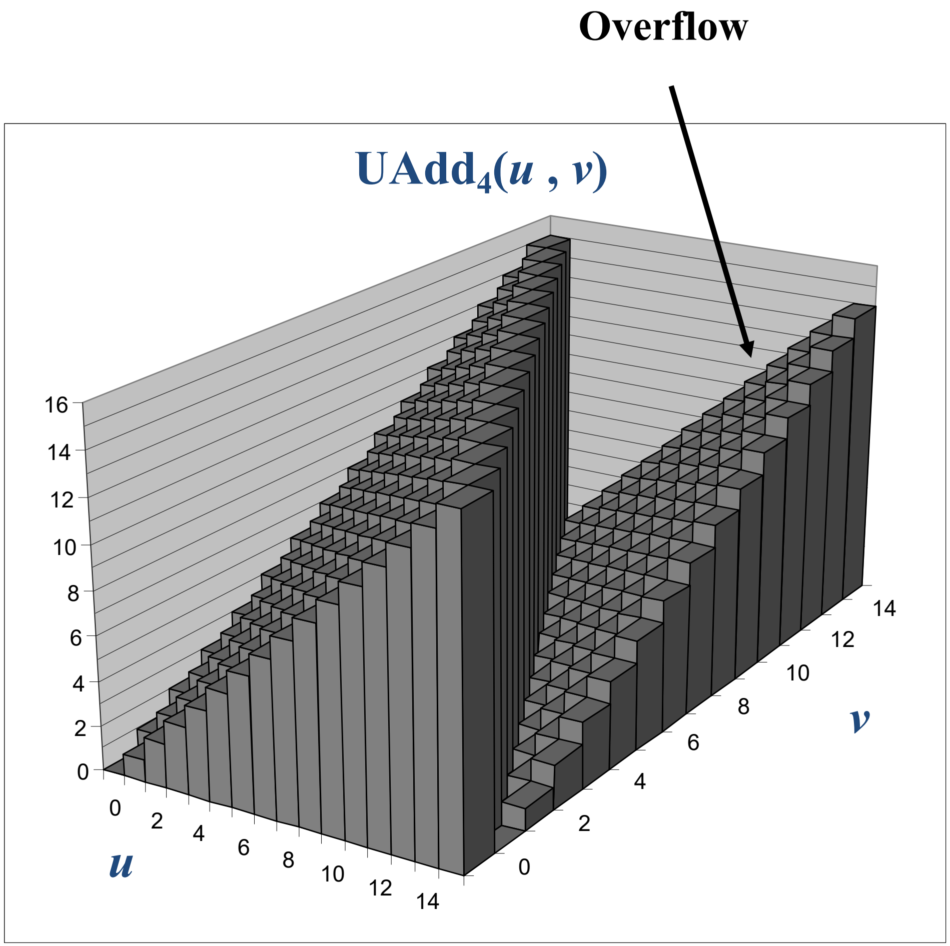
$\text{Add}_4(u\ ,\ v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum $\geq 2^w$
  - At most once





Negating with Complement & Increment

- **无符号数集合上的加法运算是模数加法**
- **已知无符号数x, 求其加法逆元：** $-^{u}_{w}x$

$$-^{u}_{w}x = \begin{cases} x & x = 0 \\ 2^{w} - x & x > 0 \end{cases}$$

- **运算规则：**
  – x的加法逆元的位模式为：对x的位模式按位取反加1

Mathematical Properties

- **无符号加法运算(模2^w)构成 阿贝尔群(*Abelian Group*)**
  - Closed under addition (封闭性)

    $0 \le UAdd_w(u, v) \le 2^w - 1$
  - Associative (结合律)

    $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$
  - 0 is additive identity (单位元)

    $UAdd_w(u, 0) = u$
  - Every element has additive inverse (加法逆元)
    - Let $UComp_w(u) = 2^w - u$

    $UAdd_w(u, UComp_w(u)) = 0$
  - Commutative (交换律)

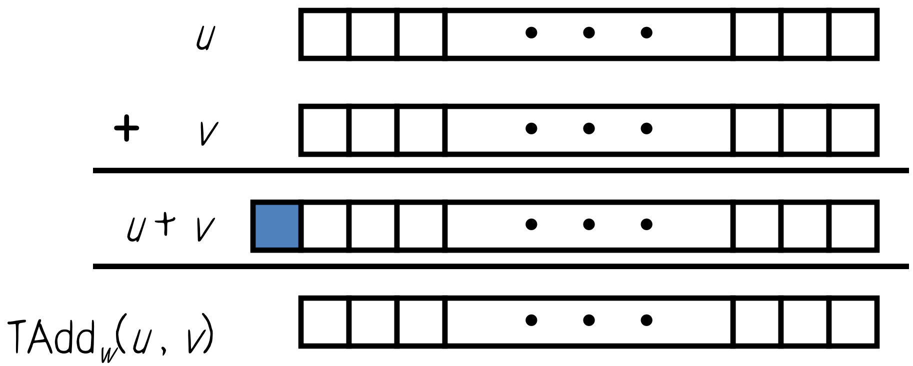
    UAddw(u , v) = UAddw(v , u)

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **2的补码加法(TAdd)和无符号整数加法(UAdd)具有一致的位级行为(Bit-Level Behavior)**
  - Signed vs. unsigned addition in C:

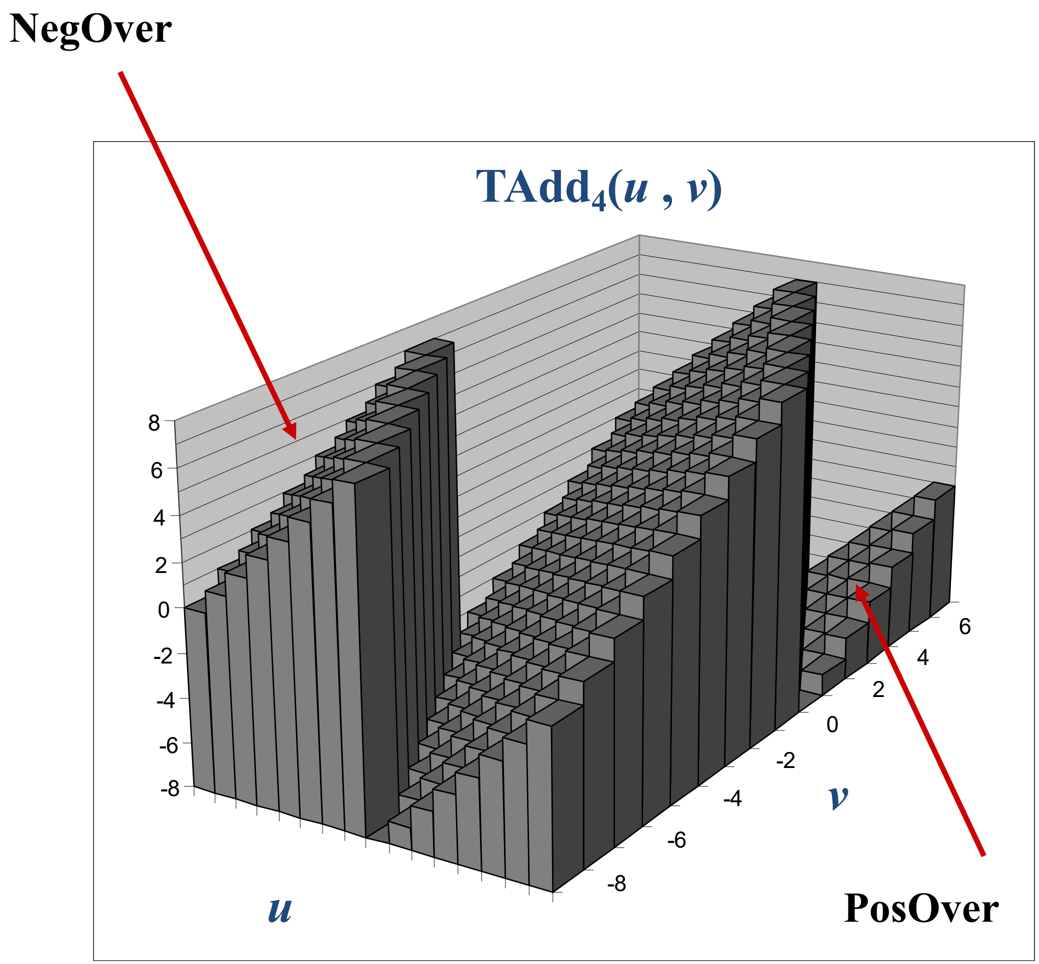
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	-66
1011 1110	BE	-66

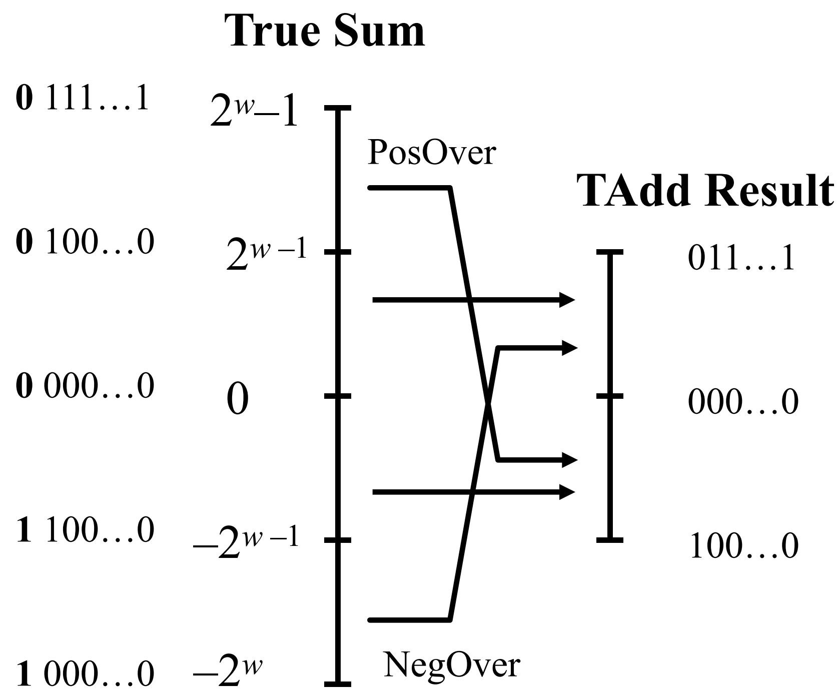
Visualizing 2's Comp. Addition

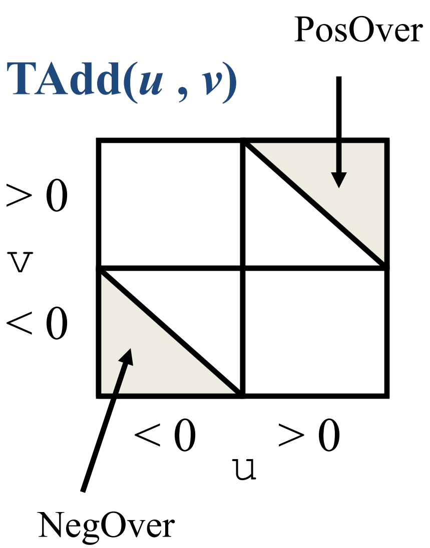
- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative (PosOver)
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes posive (NegOver)
    - At most once



Characterizing TAdd

- **所实现的功能**
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



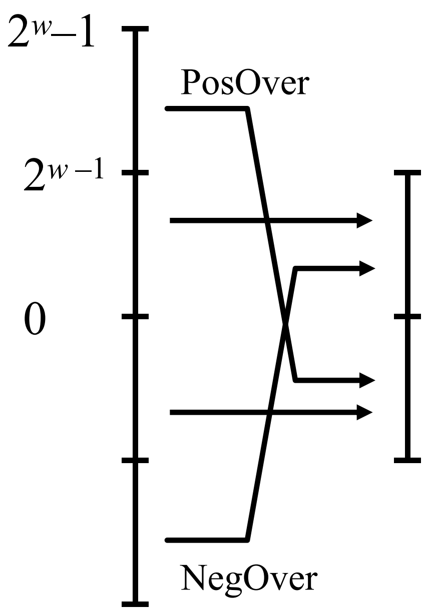


$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

Detecting 2's Comp. Overflow

- **Task**
  - Given $s = \text{TAdd}_w(u, v)$
  - Determine if $s = \text{Add}_w(u, v)$
  - Example

```
int s, u, v;
s = u + v;
```



- **Claim**
  - Overflow iff either:

$$u, v < 0, s \geq 0 \quad \text{(NegOver)}$$

$$u, v \geq 0, s < 0 \quad \text{(PosOver)}$$

```
ovf = (u<0 == v<0) && (u<0 != s<0);
```

Negating with Complement & Increment

- **求有符号整型数的加法逆元?**
- **Claim: Following Holds for 2's Complement**

  `~x + 1 == -x`

- **Complement**
  - Observation: `~x + x == 1111…11`$_2$ `== -1`

x	1	0	0	1	1	1	0	1
+ ~x	0	1	1	0	0	0	1	0
-1	1	1	1	1	1	1	1	1

- **Increment**
  - `~x + x + (-x + 1) == -1 + (-x + 1)`
  - `~x + 1 == -x`
- **Warning: Be cautious treating `int`'s as integers**
  - OK here? Tmin的加法逆元是Tmin

Comp. & Incr. Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

x=0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x=TMin

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

- **有符号数的加法逆元：** $-{}^{t}_{w}x = \begin{cases} Tmin & x = Tmin \\ -x & x > Tmin \end{cases}$
- **无符号数的加法逆元：** $-{}^{u}_{w}x = \begin{cases} x & x = 0 \\ 2^{w} - x & x > 0 \end{cases}$

Mathematical Properties of TAdd

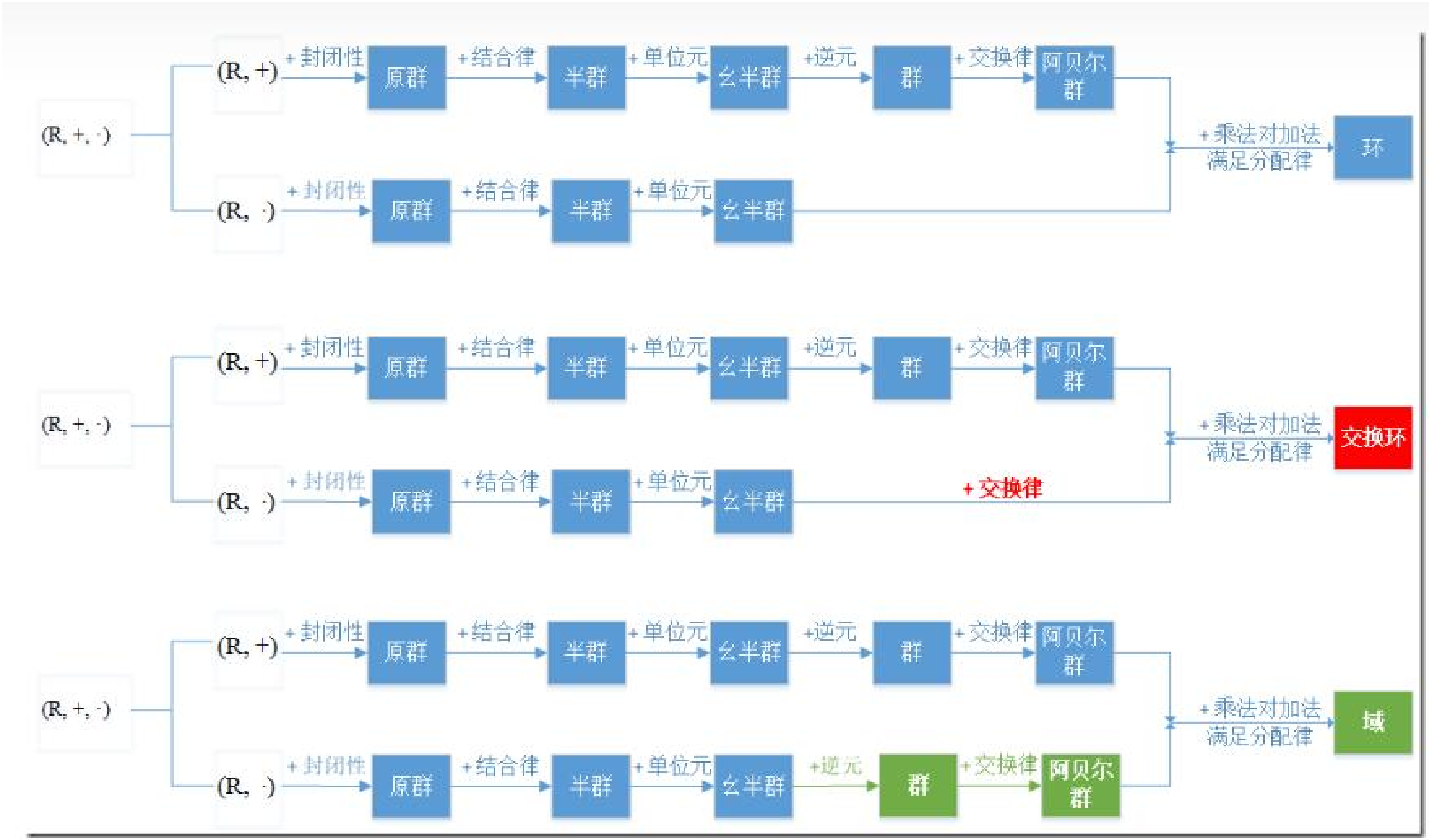
- **2的补码加法(TAdd)同构于无符号整数加法(UAdd)**
  - $\text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
    - Since both have identical bit patterns
- **2的补码加法(TAdd)构成群(阿贝尔群)**
  - Closed, Commutative, Associative, 0 is additive identity
  - **Every element has additive inverse**

    Let $\text{TComp}_w(u) = \text{U2T}(\text{UComp}_w(\text{T2U}(u))$

    $\text{TAdd}_w(u, \text{TComp}_w(u)) = 0$

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

群、环、域



Integers outline

- **Representation: unsigned and signed**
- **Conversion, casting**
- **Expanding, truncating**
- **Addition, negation, multiplication, shifting**
- **Summary**

Multiplication

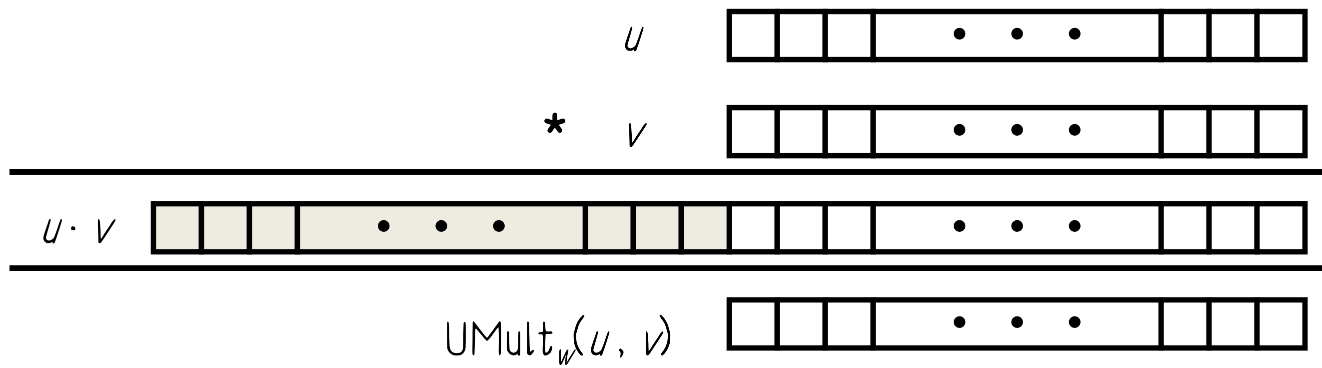
- **计算w-bit的 x, y的精确乘积**
  - Either signed or unsigned
- **实际结果将会超过w-bit的表示范围**
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = (2^w - 1) * 2^w - (2^w - 1) = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two's complement min: $x * y \ge (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $(2w\text{-}1)+1$ bits, but only for $(TMin_w)^2$
- **因此，要维持精确结果**
  - Would need to keep expanding word size with each product computed
  - Impossible in hardware (at least without limits), as all resources are finite
  - In practice, is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **标准的乘法运算$\mathrm{UMult}_w(u, v)$忽略掉高w位**
  - Ignores high order *w* bits
- **实现的是模运算**

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

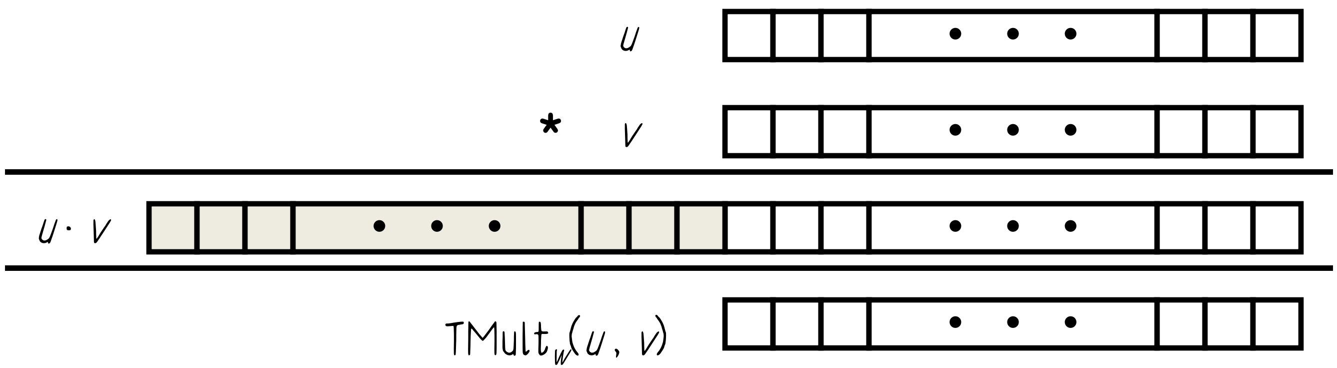
1110 1001	E9	223
* 1101 0101	* D5	* 213
1100 0001 1101 1101	C1DD	47499
1101 1101	DD	221

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **标准的乘法运算**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication (03DD vs. C1DD）
  - **Lower bits are the same**

```
            1110 1001        E9       -23
*           1101 0101    *   D5   *   -43
----------------------   ------   -------
  0000 0011 1101 1101      03DD       989
----------------------   ------   -------
            1101 1101        DD       -35
```

无符号数vs.有符号数 乘法运算规则

特性	无符号数	有符号数（补码）
核心规则	所有位均表示数值，直接进行二进制乘法	符号位参与运算，需特殊处理 (如 Booth 算法)
核心方法	转换为“移位+加法”的组合	转换为“移位+加法”的组合 (需处理符号，如Booth算法)
位扩展方式	零扩展 (Zero Extension)	符号位扩展 (Sign Extension)
移位操作	逻辑右移	算术右移
硬件实现	部分积生成和累加时高位补0	部分积生成和累加时需进行符号位扩展（例如扩展到2n位）
溢出判断	若乘积的高位部分（超出目标位宽部分）不全为0，则溢出	若乘积的高位部分不是低位部分的符号扩展，则溢出
典型指令	x86架构的 MUL指令	x86架构的 IMUL指令

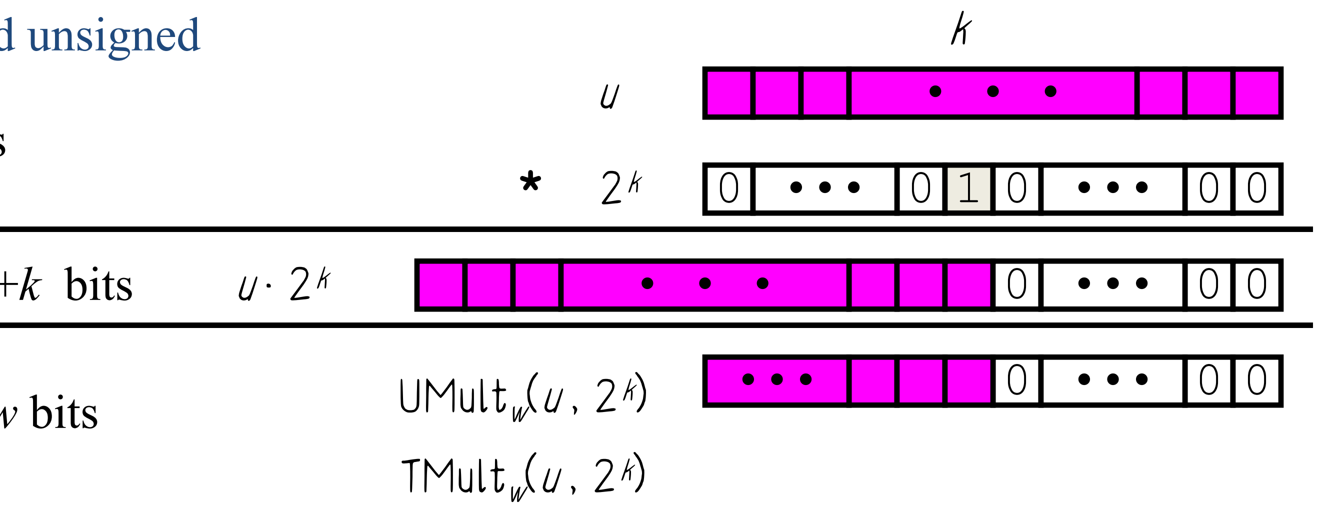
Power-of-2 Multiply with Shift

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

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



- **Examples**
  - `u << 3 == u * 8`
  - `(u << 5) - (u << 3) == u * 24`
  - Most machines shift and add much faster than multiply
    - Compiler generates this code automatically

Compiled Multiplication Code

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled Arithmetic Operations

```
  leal (%eax,%eax,2), %eax
  sall $2, %eax
```

Explanation

```
  t <- x+x*2
  return t << 2;
```

C compiler automatically generates shift/add code when multiplying by constant

Unsigned vs. Signed Multiplication

- **有符号整数补码表示转换为无符号整数运算**

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

  - Truncates product to w-bit number $up = \mathrm{UMult}_w(ux, uy)$
  - Modular arithmetic: $up = ux \cdot uy \mod 2^w$

- **有符号整数直接进行补码乘法运算**

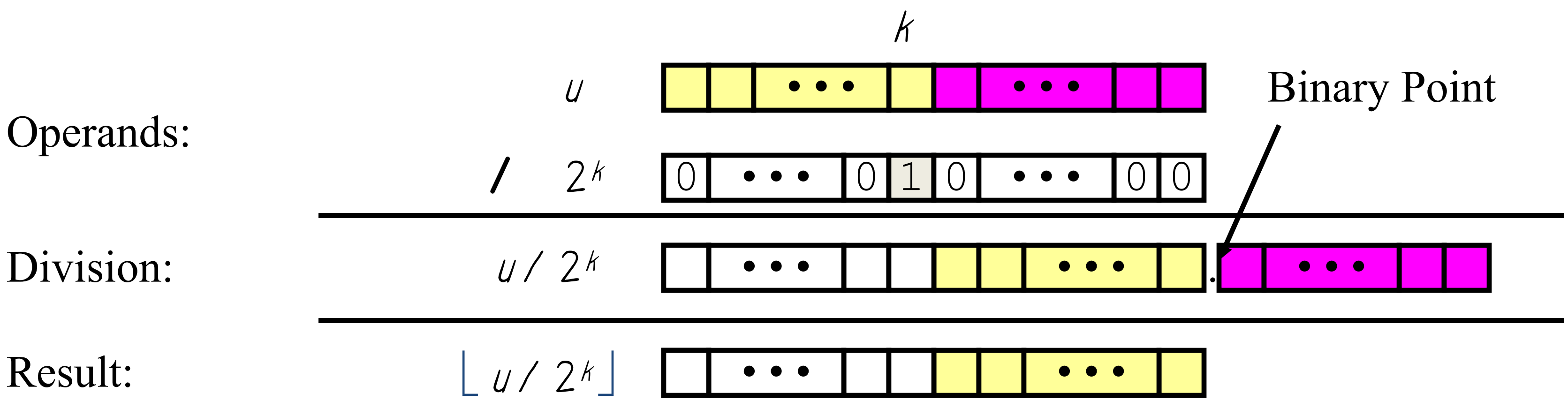
```
int x, y;
int p = x * y;
```

  - Compute exact product of two w-bit numbers x, y
  - Truncate result to w-bit number $p = \mathrm{TMult}_w(x, y)$

- **两者的关系**
  - **Signed multiplication gives same bit-level result as unsigned**
  - `up == (unsigned) p`

Unsigned Power-of-2 Divide with Shift

- **无符号整数除以2^k的结果**
  - `u >> k` gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
  shrl $3, %eax
```

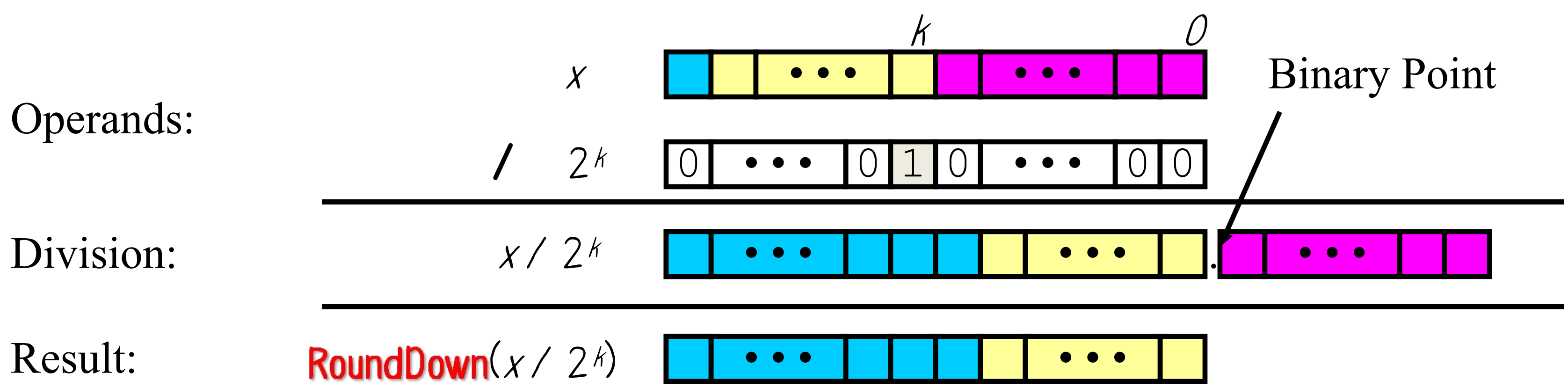
Explanation

```
  # Logical shift
  return x >> 3;
```

- **无符号整数除以2的幂可以用逻辑移位实现**
- **Java 中 逻辑右移运算符为：>>>**

Signed Power-of-2 Divide with Shift

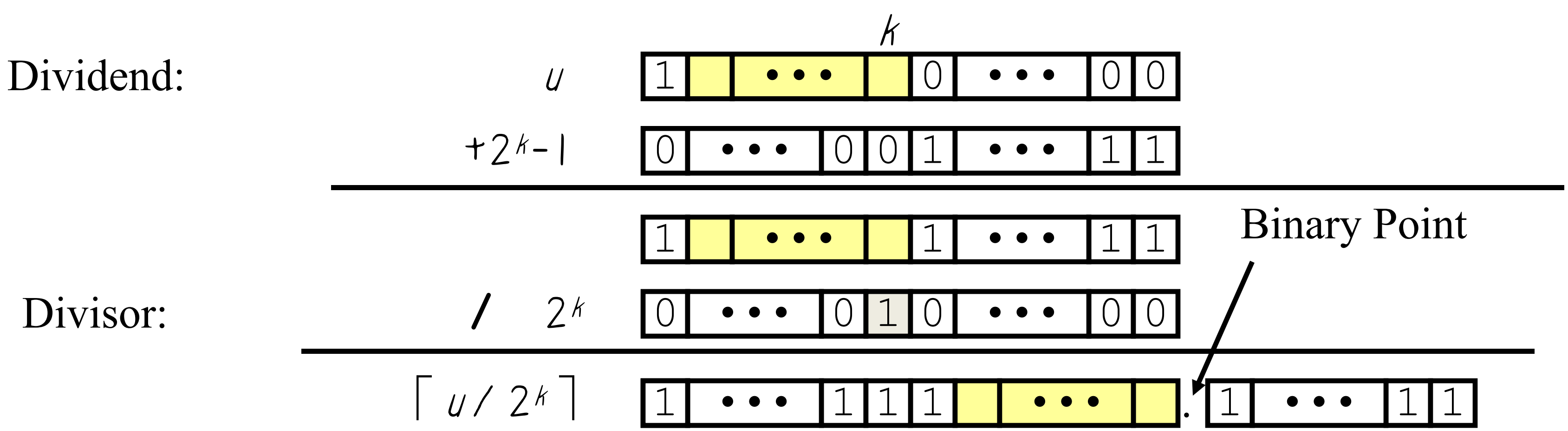
- **有符号整数除以**2^k
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - **Rounds wrong direction when `x < 0`**



	Division	Computed	Hex	Binary
`y`	-15213	-15213	C4 93	11000100 10010011
`y >> 1`	-7606.5	-7607	E2 49	**1**1100010 01001001
`y >> 4`	-950.8125	-951	FC 49	**1111**1100 01001001
`y >> 8`	-59.4257813	-60	FF C4	**11111111** 11000100

Round-toward-0 Divide（向0取整)

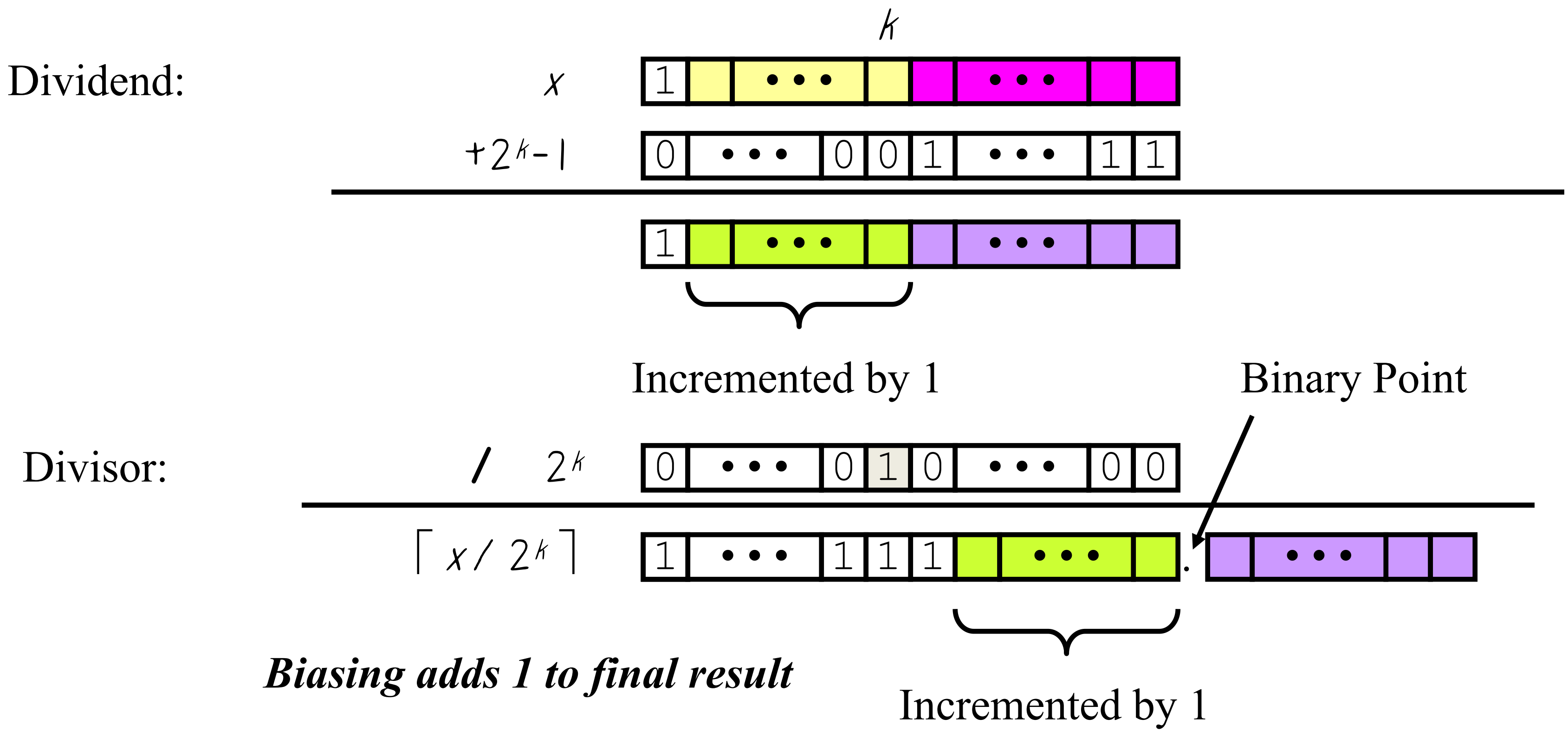
- **负数除以*2^k*的结果修正**
  - Want ⌈ x / 2^k ⌉ (**Round Toward 0**)
  - Compute as ⌊ (x+2^k−1) / 2^k ⌋
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0
- **Case 1: No rounding**



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
  testl %eax, %eax
  js   L4
L3:
  sarl $3, %eax
  ret
L4:
  addl $7, %eax
  jmp  L3
```

Explanation

```
  if x < 0
    x += 7;
  # Arithmetic shift
  return x >> 3;
```

Arithmetic: Basic Rules

- **加法:**
  - **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 2^w (result in proper range)**
    - **Mathematical addition + possible addition or subtraction of 2^w**
- **乘法:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - **Signed: modified multiplication mod 2^w (result in proper range)**

Arithmetic: Basic Rules

- **无符号整型和有符号整型运算是同构环(Isomorphic rings)。**
  - 可通过强制类型转换完成算术运算(isomorphism = casting)
- **左移操作**
  - **Unsigned/signed: multiplication by 2^k**
  - **Always logical shift**
- **右移操作**
  - **Unsigned: logical shift, div (division + round to zero) by 2^k**
  - **Signed: arithmetic shift**
    - **Positive numbers: div (division + round to zero) by 2^k**
    - **Negative numbers: div (division + round away from zero) by 2^k**

    Use biasing to fix

Properties of Unsigned Arithmetic

- **无符号乘法和加法构成交换环(Commutative Ring)**
  - **Closed under multiplication**

    $0 \le \mathbf{UMult}_w(u, v) \le 2^w - 1$
  - **Addition is commutative group**
  - **Multiplication Commutative**

    $\mathbf{UMult}_w(u, v) = \mathbf{UMult}_w(v, u)$
  - **Multiplication is Associative**

    $\mathbf{UMult}_w(t, \mathbf{UMult}_w(u, v)) = \mathbf{UMult}_w(\mathbf{UMult}_w(t, u), v)$
  - **1 is multiplicative identity**

    $\mathbf{UMult}_w(u, 1) = u$
  - **Multiplication distributes over addition**

    $\mathbf{UMult}_w(t, \mathbf{UAdd}_w(u, v)) = \mathbf{UAdd}_w(\mathbf{UMult}_w(t, u), \mathbf{UMult}_w(t, v))$

Properties of Two's Comp. Arithmetic

- **2的补码运算（加、乘）和无符号整型运算（加、乘）在位级上是一致的**
  - Unsigned multiplication and addition
    - Truncating to w bits
  - Two’s complement multiplication and addition
    - Truncating to w bits
- **2的补码加、乘运算和无符号加、乘运算都构成交换环**
  - Isomorphic to ring of integers mod 2^w

- **与数学上的整型加乘运算的比较**
  - Both are rings
  - Integers（数学） obey ordering properties, e.g.,

    $u > 0 \quad \Rightarrow \quad u + v > v$

    $u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$
  - These properties are not obeyed by two’s comp. arithmetic

    $TMax + 1 \quad == \quad TMin$

    15213 * 30426 == -10030 (16-bit words)

Why Should I Use Unsigned?

- **不要在没有理解含义的情况下使用Unsigned类型**

  – Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
a[i] += a[i+1];
```

  – Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
...
```

Counting Down with Unsigned

- **使用unsigned型变量作为循环索引的正确方法**

```
unsigned i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```

- **See Robert Seacord, Secure Coding in C and C++**
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - **0 – 1 → UMax**
- **更好的方式**

```
typedef long unsigned int  size_t
```

```
size_t i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```

  - Data type size_t defined as unsigned value with length = word size

Why Should I Use Unsigned? (cont.)

- **执行模运算时使用无符号类型**
  - Multiprecision arithmetic
- **使用bits表示集合时使用无符号整型**
  - Logical right shift, no sign extension
- **系统类程序编程时使用无符号整型**
  - Bit masks, device commands,…

C Puzzle Answers

- 假设机器字长32位，采用2的补码表示有符号整型数x
- *TMin* 在很多情况下都是一个很好的反例

❑ `x < 0`	$\Rightarrow$	`((x*2) < 0)`	**False:**	*TMin*
❑ `ux >= 0`			**True:**	$0 = UMin$
❑ `x & 7 == 7`	$\Rightarrow$	`(x<<30) < 0`	**True:**	$x_1 = 1$
❑ `ux > -1`			**False:**	0
❑ `x > y`	$\Rightarrow$	`-x < -y`	**False:**	`-1`, *TMin*
❑ `x * x >= 0`			**False:**	`30426`
❑ `x > 0 && y > 0`	$\Rightarrow$	`x + y > 0`	**False:**	*TMax*, *TMax*
❑ `x >= 0`	$\Rightarrow$	`-x <= 0`	**True:**	$-TMax < 0$
❑ `x <= 0`	$\Rightarrow$	`-x >= 0`	**False:**	*TMin*

Summary

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - **Representation: unsigned and signed**
  - **Conversion, casting**
  - **Expanding, truncating**
  - **Addition, negation, multiplication, shifting**
- **Summary**

Floating Point

- **Background: Fractional binary numbers**
- **IEEE floating point standard: Definition**
- **Example and properties**
- **Rounding, addition, multiplication**
- **Floating point in C**
- **Summary**

Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

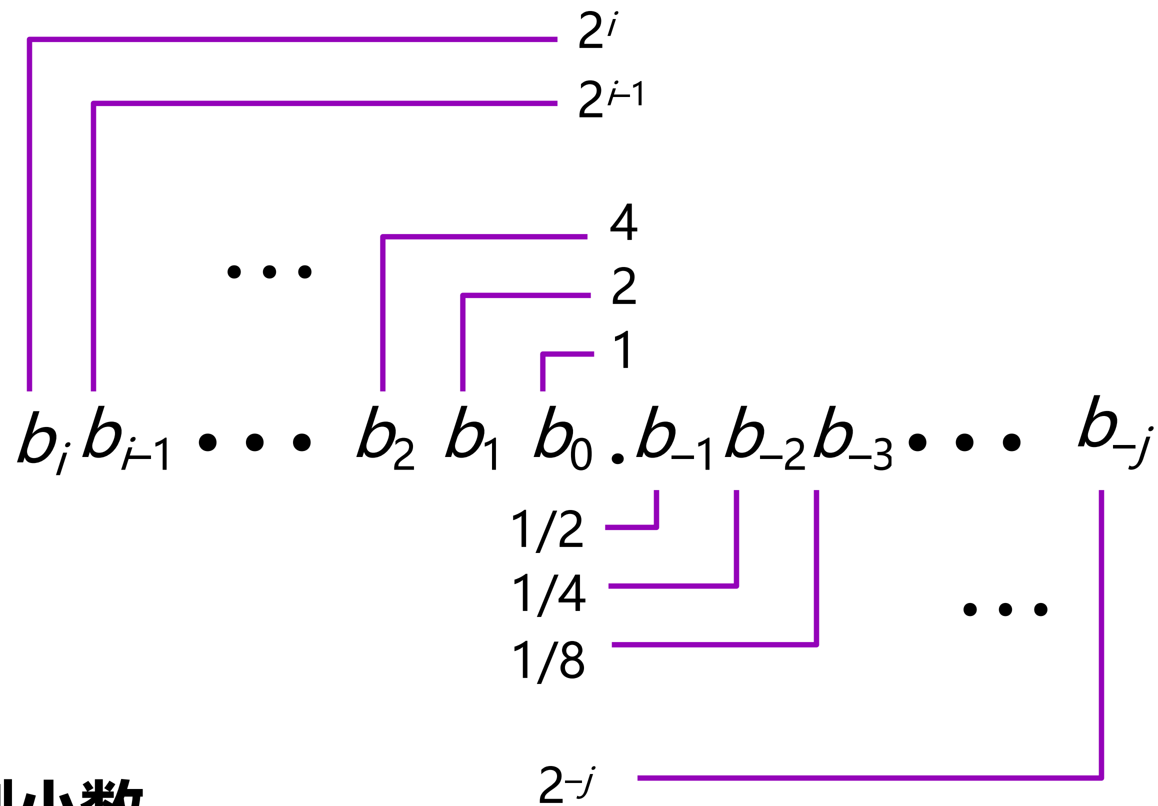
Assume neither d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

分数的二进制表示

- **What is 1011.1012?**

分数的二进制表示



- **二进制小数**
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: $$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Frac. Binary Number Examples

- **Value** **Representation**

Value	Representation	
5 + 3/4	101.11_2	= 4+1+1/2+1/4
2 + 7/8	10.111_2	= 2+1/2+1/4+1/8
63/64	0.111111_2	= 1/2+1/4+1/8+1/16+1/32+1/64

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation #1**
  - 仅能精确表示形如$x/2^k$ *的有理数*
    - Other numbers have repeating bit representations
- **Value** **Representation**
  - 1/3 `0.0101010101[01]...`$_2$
  - 1/5 `0.001100110011[0011]...`$_2$
  - 1/10 `0.0001100110011[0011]...`$_2$

- **Limitation #2**
  - 在w位宽中，仅设置一个小数点
    - Limited range of numbers (very small values? very large?)

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
  - Some CPUs don't implement IEEE 754 in full

  e.g., early GPUs, Cell BE processor
- **Driven by Numerical Concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make go fast in hardware
    - Numerical analysts predominated over hardware types in defining standard

IEEE 754-2019

Table 3.1—Relationships between different specification levels for a particular format

Level 1	$\{-\infty \ldots\ \mathbf{0}\ \ldots +\infty\}$	Extended real numbers.
many-to-one ↓	*rounding*	↑ projection (except for NaN)
Level 2	$\{-\infty \ldots -\mathbf{0}\} \cup \{+\mathbf{0} \ldots +\infty\} \cup$ **NaN**	Floating-point data—an algebraically closed system.
one-to-many ↓	*representation specification*	↑ many-to-one
Level 3	(***sign, exponent, significand***) $\cup \{-\infty, +\infty\} \cup$ **qNaN** $\cup$ **sNaN**	Representations of floating-point data.
one-to-many ↓	*encoding for representations of floating-point data*	↑ many-to-one
Level 4	**0111000…**	Bit strings.

The mathematical structure underpinning the arithmetic in this standard is the extended reals, that is, the set of real numbers together with positive and negative infinity. For a given format, the process of *rounding* (see Clause 4) maps an extended real number to a *floating-point number* included in that format. A *floating-point datum*, which can be a signed zero, finite non-zero number, signed infinity, or a NaN (not-a-number), can be mapped to one or more *representations of floating-point data* in a format.

Floating Point Representation

- **数值表示：** $(-1)^s\ M\ \ 2^E$
  - Sign bit s determines whether number is negative or positive
  - Significand(尾数) M normally a fractional value in range [1.0,2.0).
  - Exponent(阶码) E weights value by power of two

- **编码格式**
  – MSB is sign bit
  – `exp` field encodes E
  – `frac` field encodes M

s	exp	frac

Floating Point Precisions

- **单精度Single precision: 32 bits ≈7 decimal digits, $10^{\pm 38}$**

s	exp	frac
1	8-bits	23-bits

- **双精度Double Precision：64bits ≈16 decimal digits, $10^{\pm 308}$**

s	exp	frac
1	11-bits	52-bits

- **扩展精度: 80 bits (Intel only)**

s	exp	frac
1	15-bits	63 or 64-bits

- **其他格式: half precision (FP16) , quad precision, FP8**

AI领域的主流浮点数格式

格式	符号位	指数位	尾数位	总位数	适用范围
FP64	1	11	52	64	高精度计算
FP32	1	8	23	32	通用训练、高精度计算
FP16	1	5	10	16	混合精度训练、移动端推理
FP8 E4M3	1	4	3	8	LLM训练、高性能计算
FP8 E5M2	1	5	2	8	LLM训练、高性能计算
TF32	1	8	10	实际19位	NVIDIA GPU加速的深度学习
BF16	1	8	7	16	大模型训练（如GPT）

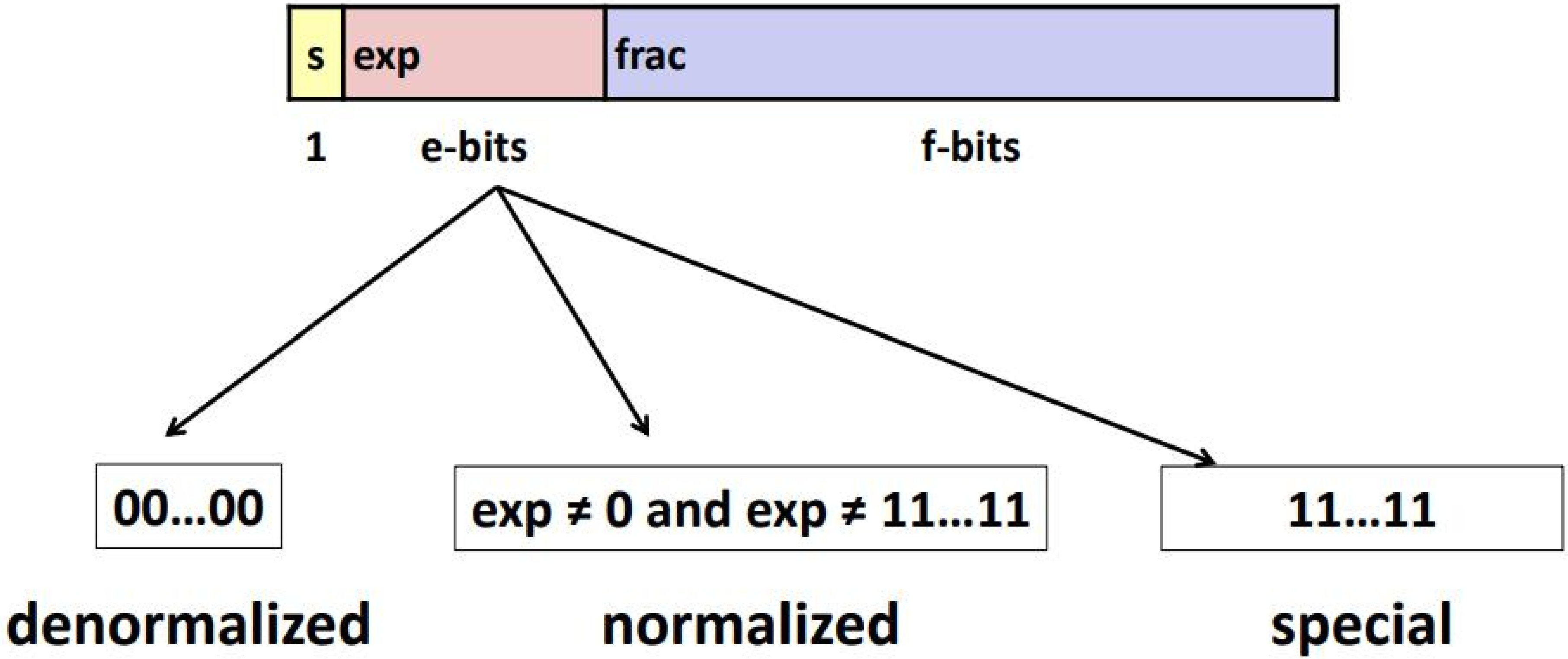
FP8在深度学习模型中的推理精度

网络	Float32 推理精度	INT8 推理精度	HFP8 推理精度	QFP8 推理精度
AlexNet	56.518	56.116	54.300	56.474
DenseNet121	74.434	74.104	70.260	74.322
GoogleNet	69.186	68.812	67.208	68.924
Inception_V3	77.488	76.944	73.206	77.246
MobileNet_V3	75.442	53.286	70.320	74.020
Resnet101	77.374	76.896	72.146	77.112
Resnet101_V2	74.456	74.004	73.992	74.344
Resnet50	76.130	75.606	72.198	76.024
Resnet50_v2	73.298	72.974	71.972	73.152
VGG16	71.592	71.514	68.318	71.448
VGG16_BN	73.360	73.180	64.666	73.256

不同Float8格式的模型推理精度

网络	Float32 推理精度	1-2-5 推理精度	1-3-4 推理精度	1-4-3 推理精度	1-5-2 推理精度
AlexNet	56.518	56.142	56.474	56.000	55.176
DenseNet121	74.434	74.284	74.322	73.984	72.198
GoogleNet	69.186	68.868	68.924	68.378	62.388
Inception_V3	77.488	77.272	77.246	76.244	72.386
MobileNet_V3	75.442	57.534	74.020	72.944	52.326
Resnet101	77.374	77.148	77.112	76.686	74.956
Resnet101_V2	74.456	74.158	74.344	74.016	72.528
Resnet50	76.130	75.802	76.024	75.376	73.408
Resnet50_v2	73.298	73.108	73.152	72.844	68.238
VGG16	71.592	71.480	71.448	71.084	69.912
VGG16_BN	73.360	73.330	73.256	72.974	71.070

Three “kinds” of floating point numbers



“Normalized” Numeric Values

- **规格化浮点数表示的情况：**
  - **Condition: exp ≠ 000…0 and exp ≠ 111…1**
- **指数E(有符号整数）编码为增加了偏置值（移码）的非负整数(exp)**
  - **Exponent coded as *biased* value：** $E = exp - Bias$
  - *exp* : **unsigned value** denoted by exp
  - *Bias* : Bias value
    - Single precision: 127 (*exp*: 1…254 → *E*: -126…127) (阶码8位）
    - Double precision: 1023 (*exp*: 1…2046 → *E*: -1022…1023) （阶码11位）
    - in general: ***Bias* = $2^{e-1} - 1$**, where e is number of exponent bits
- **尾数编码解释为隐含以1开头的小数表示**
  - **Significand coded with implied leading 1：** $M = 1.xxx…x_2$
  - xxx…x: bits of frac
  - Minimum when 000…0 ($M = 1.0$)
  - Maximum when 111…1 ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for “free”

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

Normalized Encoding Example

- **Value:** `Float F = 15213.0;`
  - $15213_{10} = 11101101101101_2 = 1.1101101101101_2 * 2^{13}$

- **Significand**

  $M = 1.\underline{1101101101101}_2$

  frac= $\underline{1101101101101}0000000000_2$ (23位)

- **Exponent**

  $E = 13$

  $Bias = 127$

  $exp = 140 = 10001100_2$

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

$$E = exp - Bias$$

Floating Point Representation (Class 02):

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
140:	100	0110	0					
15213:			*1*110	1101	1011	01		

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

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

$E = 1 - Bias$

- **非规格化浮点数表示的情况**
  - Condition： `exp` = `000…0`
- **阶码和尾数部分的解释**
  - Exponent value $E = -Bias + 1$
  - **Significand coded with implied leading 0**： $M = \texttt{0.xxx...x}_2$
    - `xxx…x`: bits of `frac`
- **分为两种情况**
  - **`exp` = `000…0`, `frac` = `000…0`**
    - Represents value 0
    - Note that have distinct values +0 and –0
  - **`exp` = `000…0`, `frac` ≠ `000…0`**
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - "Gradual underflow"

Special Values

- **特殊值的情况： exp = 111…1**
- **情形1： exp = 111…1, frac = 000…0**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- **情形2：exp = 111…1, frac ≠ 000…0**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$， $* \; 0$

C float Decoding Example

- `float: 0xC0A00000`
- `binary: 1100 0000 1010 0000 0000 0000 0000 0000`

1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

- `E = exp - Bias = 129 - 127 = 2 (decimal)`
- `S = 1 -> negative number`
- `M = 1.010 0000 0000 0000 0000 0000`
  `= 1 + 1/4 = 1.25`
- $v = (-1)^{s}\ M\ 2^{E} = (-1)^{1} * 1.25 * 2^{2} = -5$

$$v = (-1)^{s}\, M\, 2^{E}$$
$$E = \text{exp} - Bias$$

C float Decoding Example #2

- `float: 0x001C0000`
- `binary: 0000 0000 0001 1100 0000 0000 0000 0000`

0	0000 0000	001 1100 0000 0000 0000 0000
1	8-bits	23-bits

- E = 1- Bias = 1 – 127 = -126 (decimal)
- S = 0 -> positive number
- M = 0.001 1100 0000 0000 0000 0000

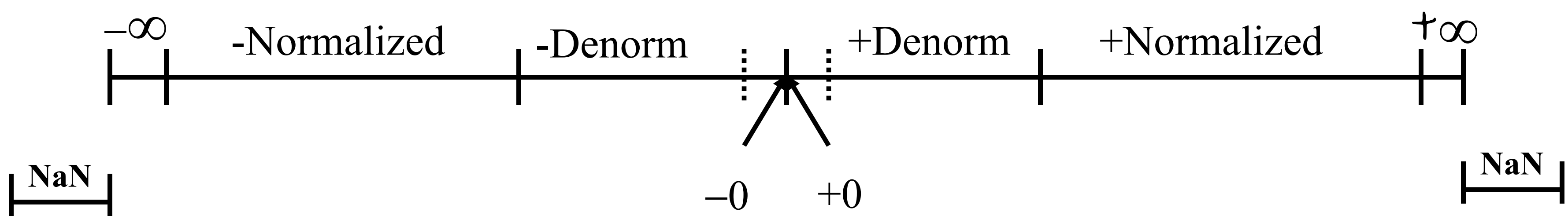
  $= 1/8 + 1/16 + 1/32 = 7/32 = 7*2^{-5}$
- $v = (-1)^{s} M\, 2^{E} = (-1)^{0} * 7*2^{-5} * 2^{-126} = 7*2^{-131}$

  $\approx 2.571393892 \times 10^{-39}$

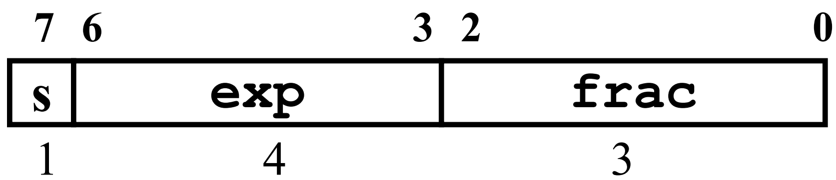
$$v = (-1)^{s} M\, 2^{E}$$
$$E = 1 - Bias$$

$Bias = 2^{k-1} - 1 = 127$

Summary of Floating Point Real Number Encoding



Tiny Floating Point Example



- **8-bit 浮点数表示**
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, **with a bias of 7**. $(2^{4-1}-1)$
  - the last three bits are the `frac`
- **与IEEE格式形式相同**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Values Related to the Exponent

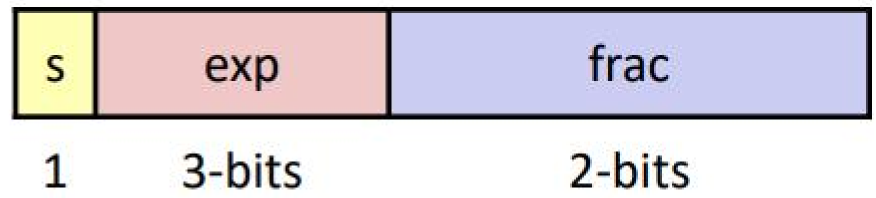
Exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, NaN)

Dynamic Range

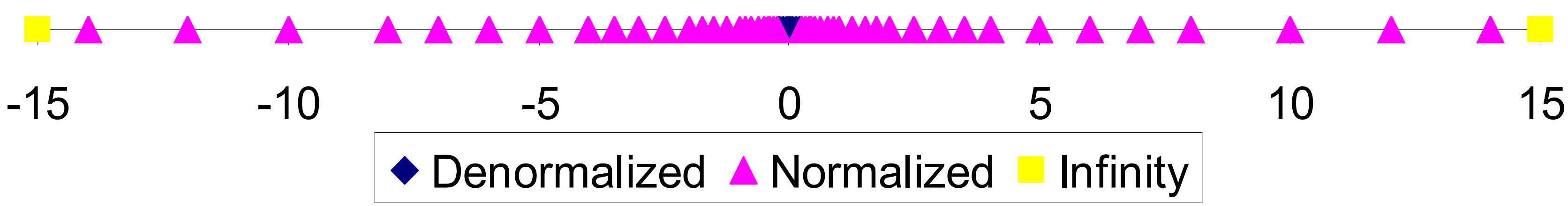
```
                 s exp  frac   E   Value

                 0 0000 000   -6   0
                 0 0000 001   -6   1/8*1/64 = 1/512     ← closest to zero
Denormalized     0 0000 010   -6   2/8*1/64 = 2/512
numbers          …
                 0 0000 110   -6   6/8*1/64 = 6/512
                 0 0000 111   -6   7/8*1/64 = 7/512     ← largest denorm
.................................................................................
                 0 0001 000   -6   8/8*1/64 = 8/512     ← smallest norm
                 0 0001 001   -6   9/8*1/64 = 9/512
                 …
                 0 0110 110   -1   14/8*1/2 = 14/16
                 0 0110 111   -1   15/8*1/2 = 15/16     ← closest to 1 below
Normalized       0 0111 000    0   8/8*1    = 1
numbers          0 0111 001    0   9/8*1    = 9/8      ← closest to 1 above
                 0 0111 010    0   10/8*1   = 10/8
                 …
                 0 1110 110    7   14/8*128 = 224
                 0 1110 111    7   15/8*128 = 240      ← largest norm
.................................................................................
                 0 1111 000   n/a  inf
```

Dynamic Range

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

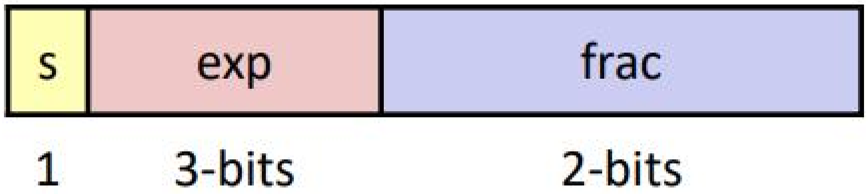


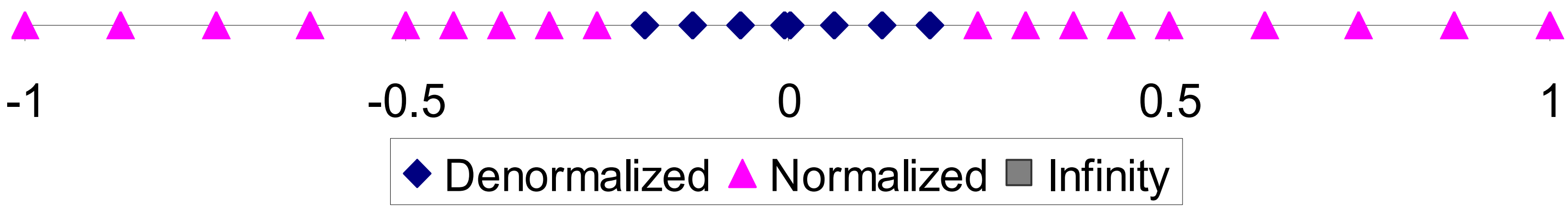
- **Notice: the distribution gets denser towards 0.**



Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Interesting Numbers

Description	exp	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}			
– Double ≈ 1.8×10^{308}			

Special Properties of Encoding

- **浮点数与整型数零的表示相同**
  - All bits = 0
- **大多数情况下无符号整型数比较规则适用于浮点数**
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs problematic
    - Will be greater than any other values?
    - What should comparison yield? The answer is complicated.
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point

- **Background: Fractional binary numbers**
- **IEEE floating point standard: Definition**
- **Example and properties**
- **Rounding, addition, multiplication**
- **Floating point in C**
- **Summary**

Floating Point Operations: Basic Idea

- $x +_f y = \mathrm{Round}(x + y)$
- $x \times_f y = \mathrm{Round}(x \times y)$
- **基本思路**
  - 首先计算精确结果
  - 使其符合所需的精度要求 （舍入规则）
    - 指数过大可能会产生溢出
    - 可能需要舍入操作以适配尾数字段

Floating Point Operations

- **基本思路**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`
- **舍入方式 (illustrate with $ rounding)**

	\$1.40	**\$1.60**	**\$1.50**	**\$2.50**	**\$-1.50**
– Zero	\$1	\$1	\$1	\$2	–\$1
– Round down (-∞)	\$1	\$1	\$1	\$2	–\$2
– Round up (+∞)	\$2	\$2	\$2	\$3	–\$1
– Nearest Even (default)	\$1	\$2	**\$2**	**\$2**	–\$2

Note:

1. **Round down: rounded result is close to but no greater than true result.**
2. **Round up: rounded result is close to but no less than true result.**

Closer Look at Round-To-Even

- **IEEE 754默认的舍入方式：Round-To-Even**
  - Hard to get any other kind without dropping into assembly
    - C99 has support for rounding mode management
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated
- **Round-To-Even （四舍六入五成双）**
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- **二进制小数**
  - "Even" when least significant bit is 0
  - **Half way when bits to right of rounding position = $100..._2$**
- **例如**
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Rounding

Guard bit: LSB of result

1.BBGRXXX

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **向上舍入(Round up)的条件**
  - Round = 1, Sticky = 1 ➙ > 0.5
  - Guard = 1, Round = 1, Sticky = 0 ➙ Round to even

Fraction	GRS	Incr?	Rounded
1.0000000	000	N	1.000
1.1010000	100	N	1.101
1.0001000	010	N	1.000
1.0011000	110	Y	1.010
1.0001010	011	Y	1.001
1.1111100	111	Y	10.000

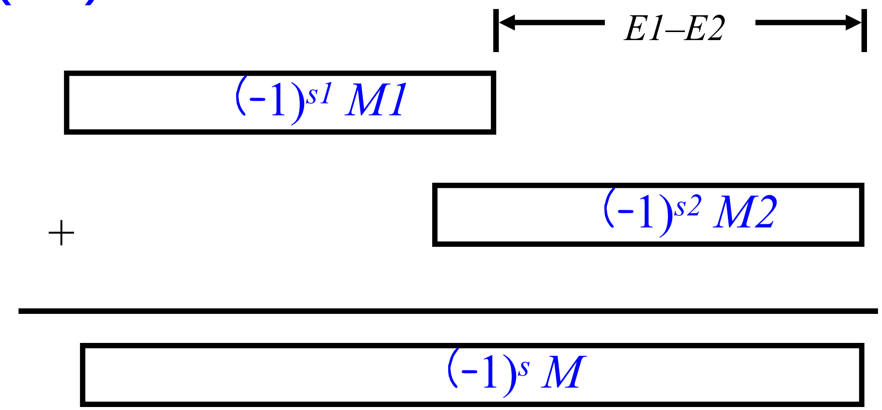
FP Multiplication

- **两个操作数:** $(-1)^{s1}\ M1\ \ 2^{E1} \times (-1)^{s2}\ M2\ \ 2^{E2}$
- **具体运算结果:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Significand *M*: $M1 * M2$
  - Exponent *E*: $E1 + E2$
- **结果调整**
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision
- **实现工作量 --尾数相乘**
  - Biggest chore is multiplying significands

4 bit significand: $1.010*2^2 \times 1.110*2^3 = 10.0011*2^5$
$= 1.00011*2^6 = 1.001*2^6$

FP Addition

- **两个操作数：** $(-1)^{s1}\ M1\ \ 2^{E1} + (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume *E1* > *E2*
- **具体运算结果:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*



- **结果调整**
  - 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

$$1.010*2^{2} + 1.110*2^{3} = (0.1010 + 1.1100)*2^{3}$$
$$= 10.0110 * 2^{3} = 1.00110 * 2^{4} = 1.010 * 2^{4}$$

Mathematical Properties of FP Add

- **是否构成 阿贝尔群(Abelian Group)**
  - Closed under addition? YES
    - But may generate infinity or NaN
  - 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
    - Except for infinities & NaNs
- **是否满足单调性(Monotonicity)**
  - $a \geq b \Rightarrow a+c \geq b+c$? ALMOST
    - Except for infinities & NaNs

Math. Properties of FP Mult

- **是否构成交换环(Commutative Ring)**
  - Closed under multiplication? YES
    - But may generate infinity or NaN
  - Multiplication Commutative? YES
  - **Multiplication is Associative? NO**
    - Possibility of overflow, inexactness of rounding
  - 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
- **是否满足单调性(Monotonicity)**
  - $a \geq b \ \ \& \ c \geq \ 0 \ \Rightarrow a * c \geq b * c$? ALMOST
    - Except for infinities & NaNs

Floating Point in C

- **C 支持两种精度的浮点数操作**

`float`	single precision
`double`	double precision

- **不同数据类型间的转换规则**
  - Casting between `int`, `float`, & `double` changes numeric values and bit representation
  - `Double` or `float` to `int`
    - **Truncates fractional part**
    - Like rounding toward zero
    - Not defined when out of range
      - Generally saturates to TMin
  - `int` to `double`
    - Exact conversion, as long as int has ≤ 53 bit word size
  - **`int` to `float`**
    - Will round according to rounding mode

Answers to Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

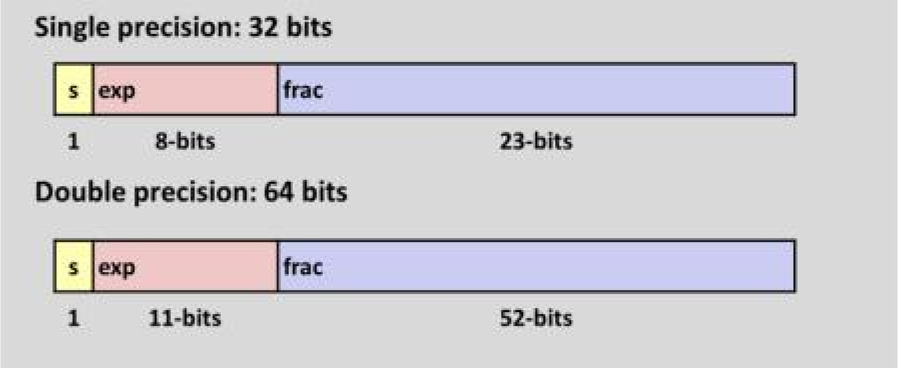
Assume neither d nor f is NAN

- **x == (int)(float) x** — **No: 24 bit significand**
- **x == (int)(double) x** — **Yes: 53 bit significand**
- **f == (float)(double) f** — **Yes: increases precision**
- **d == (float) d** — **No: loses precision**
- **f == -(-f);** — **Yes: Just change sign bit**
- **2/3 == 2/3.0** — **No: 2/3 == 0**
- **d < 0.0 → ((d*2) < 0.0)** — **Yes!**
- **d > f → -f > -d** — **Yes!**
- **d * d >= 0.0** — **Yes!**
- **(d+f)-d == f** — **No: Not associative**

Summary

- **IEEE 754标准的浮点数运算具有清晰的数学性质**
  - 我们可以不基于实现来预测其操作行为
  - As if computed with perfect precision and then rounded浮点数的表示形式为 $M \times 2^E$

- **与数学中的算术运算不同之处：**
  - **Violates associativity/distributivity**
  - **Makes life difficult for compilers & serious numerical applications programmers**



Additional Slides: Creating Floating Point Number

s	exp	frac
1	4-bits	3-bits

- **基本步骤**
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding
- **举例**
  - Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Normalize

s	exp	frac
1	4-bits	3-bits

- **基本步骤**
  - Set binary point so that numbers of form 1.xxxxx
  - Adjust all to have leading one
    - Decrement exponent as shift left

Value	*Binary*	*Fraction*	*Exponent*
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Postnormalize

- **后序规格化处理**
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	*Rounded*	*Exp*	*Adjusted*	*Numeric 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

This is important!

- **Ariane 5 在其首次发射中爆炸:-造成500万美元损失**
  - Exploded 37 seconds after liftoff，Cargo worth $500 million

- **原因：**
  - 64位浮点数转换为16位整数
    - Computed horizontal velocity as floating point number
    - **Converted to 16-bit integer**
    - Worked OK for Ariane 4
    - Overflowed for Ariane 5
    - Used same software
  - Causes rocket to get incorrect value of horizontal velocity and crash
- **爱国者导弹防御系统未命中飞毛腿- 28人死亡**
  - 系统以1/10秒为单位追踪时间
  - **将整数转换为浮点数 （大整数转换为浮点数时，会引入舍入误差）**
  - 累计舍入误差导致漂移，8小时内漂移达20%
  - 最终（1991年2月25日系统持续运行100小时后）导致距离估算误差过大，致使来袭导弹未能被拦截。

Acknowledgements

- **This course was developed and fine-tuned by Randal E. Bryant and David O'Hallaron. They wrote *The Book*!**
- **http://www.cs.cmu.edu/~./213/schedule.html**

群、环、域的定义