计算机组成与系统结构 Computer Organization & System Architecture

Huang Kejie (百人计划研究员)

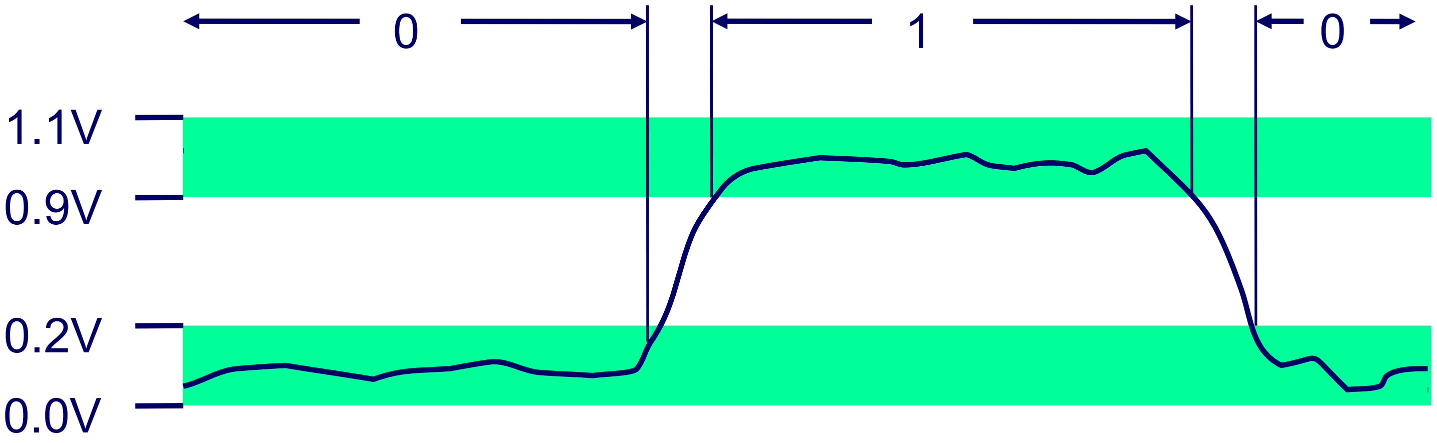
Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Count in binary

- Base 2 Number Representation
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
  - Represent $1.5213 \mathrm{X} 10^4$ as $1.1101101101101_2 \mathrm{X} 2^{13}$

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - 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
      - 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

Binary Number Property

Claim

$$1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^{w}$$

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

- $w = 0$:

  $1 = 2^0$

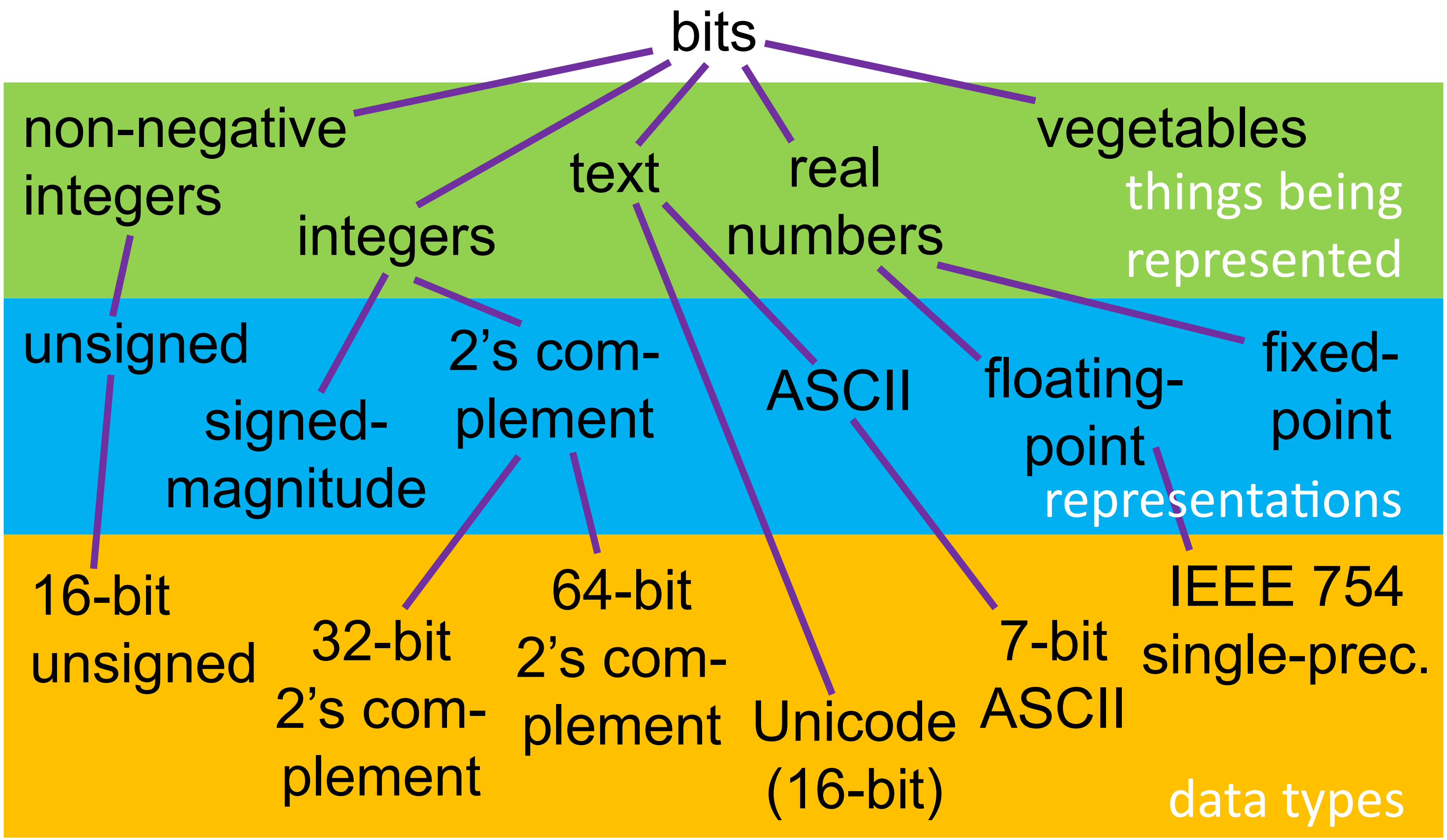
- Assume true for w-1:

  $$1 + \underbrace{1 + 2 + 4 + 8 + \ldots + 2^{w-1}}_{= \quad 2^{w}} + 2^{w} \quad = \quad 2^{w} + 2^{w} \quad = \quad 2^{w+1}$$

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
`long double`	–	–	10/16
pointer	4	8	8

Illustration of a Representation Taxonomy

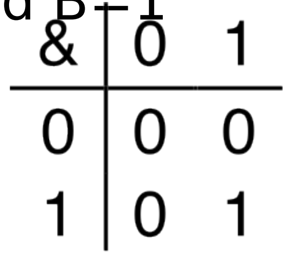


Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

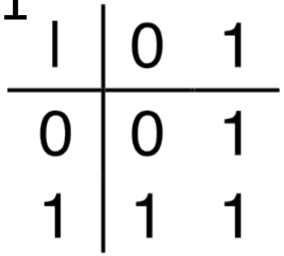
- A&B = 1 when both A=1 and B=1



&	0	1
0	0	0
1	0	1

Or

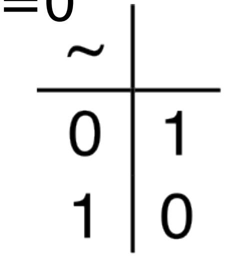
- A|B = 1 when either A=1 or B=1



|	0	1
0	0	1
1	1	1

Not

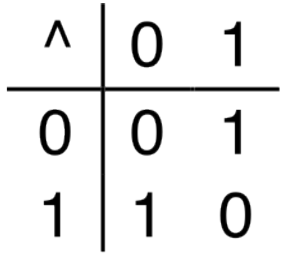
- ~A = 1 when A=0



~	
0	1
1	0

Exclusive-Or (Xor)

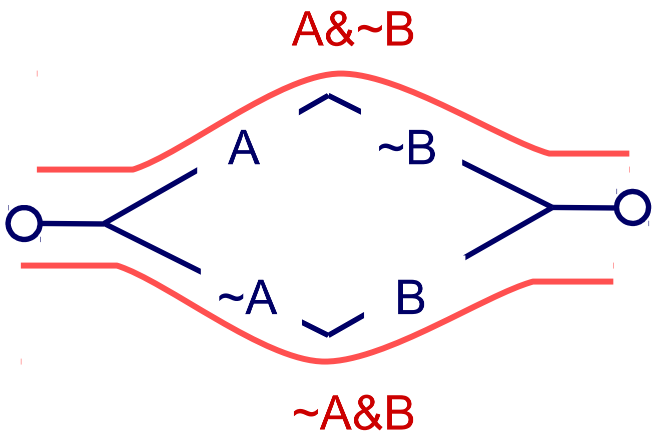
- A^B = 1 when either A=1 or B=1, but not both



^	0	1
0	0	1
1	1	0

Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

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

- All of the Properties of Boolean Algebra Apply

Example: Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001{ 0, 3, 5, 6 }
    - 76543210
    - 01010101{ 0, 2, 4, 6 }
    - 76543210
- Operations
  - & 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

- Operations &, |, ~, ^ Available in C
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (Char data type)
  - ~0x41 → 0xBE
    - ~01000001_2 → 10111110_2
  - ~0x00 → 0xFF
    - ~00000000_2 → 11111111_2
  - 0x69 & 0x55 → 0x41
    - 01101001_2 & 01010101_2 → 01000001_2
  - 0x69 | 0x55 → 0x7D
    - 01101001_2 | 01010101_2 → 01111101_2

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View s “False”
    - Anythi ero as “True”
    - Always
    - Early ter
- Examp
  - !0
  - !0
  - !!(
  - 0x
  - 0x
  - p

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in C programming

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: 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 left
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

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

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 short 2 bytes long

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

- Sign Bit
  - For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Two-complement 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

- 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

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

- Observations
  - |*TMin* | = *TMax* + 1
    - Asymmetric range
  - *Umax* = 2 * *TMax* + 1

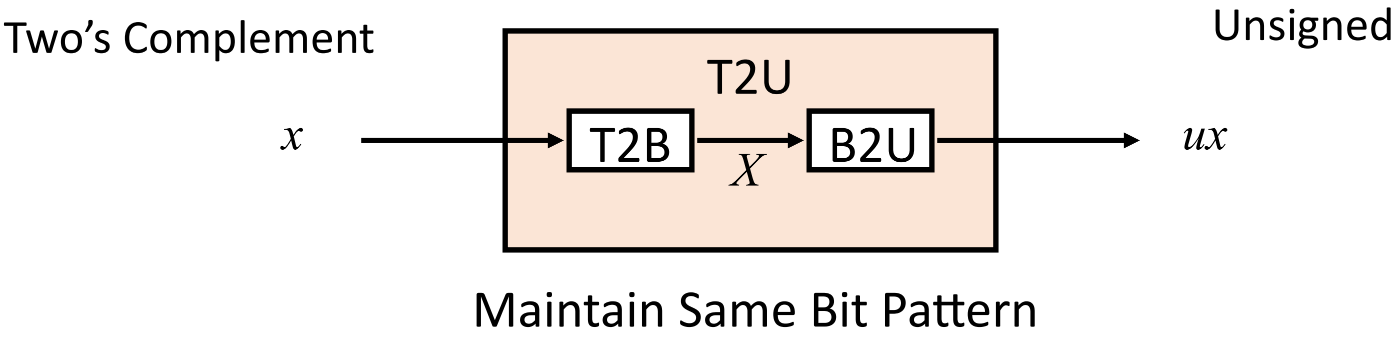
- C Programming
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

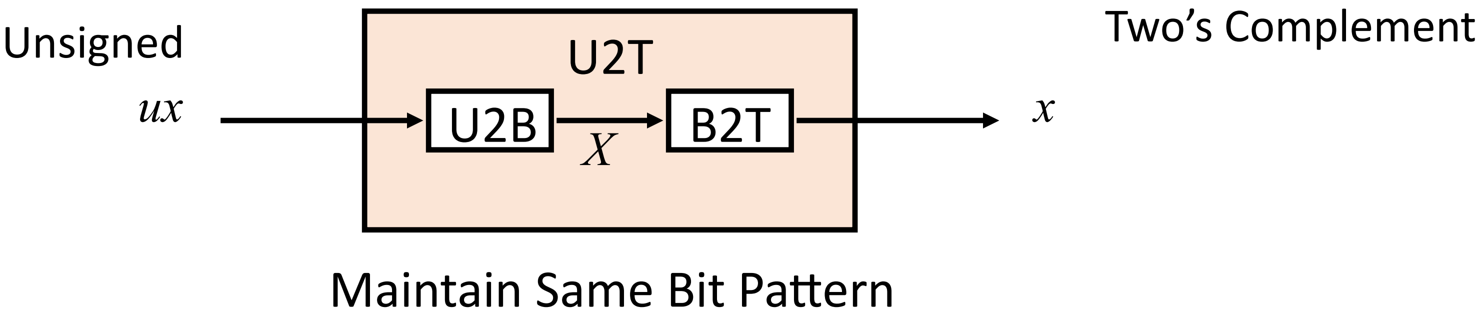
Unsigned & Signed Numeric Values

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

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- ⇒ Can Invert Mappings
  - U2B(*x*) = $\text{B2U}^{-1}(x)$
    - Bit pattern for unsigned integer
  - T2B(*x*) = $\text{B2T}^{-1}(x)$
    - Bit pattern for two’s comp integer

Mapping Between Signed & Unsigned

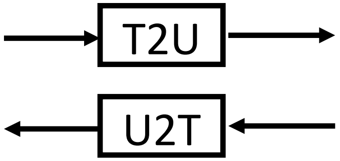




- Mappings between unsigned and two's complement numbers: Keep bit representations and reinterpret

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



T2U

U2T

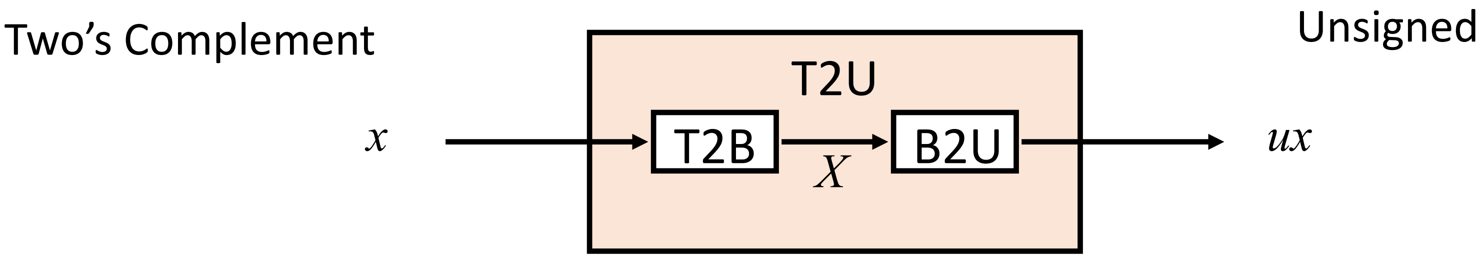
Mapping Signed ↔ Unsigned

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

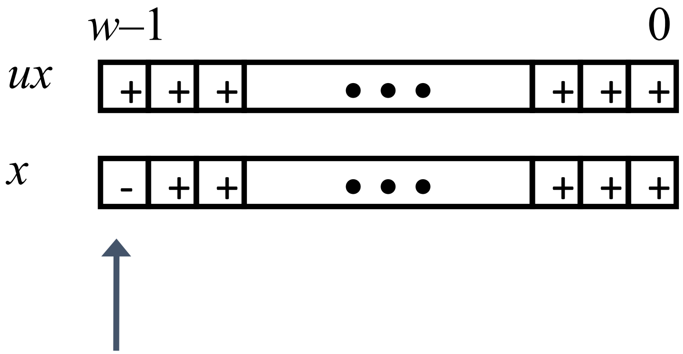
↔ =

↔ +/- 16

Relation between Signed & Unsigned

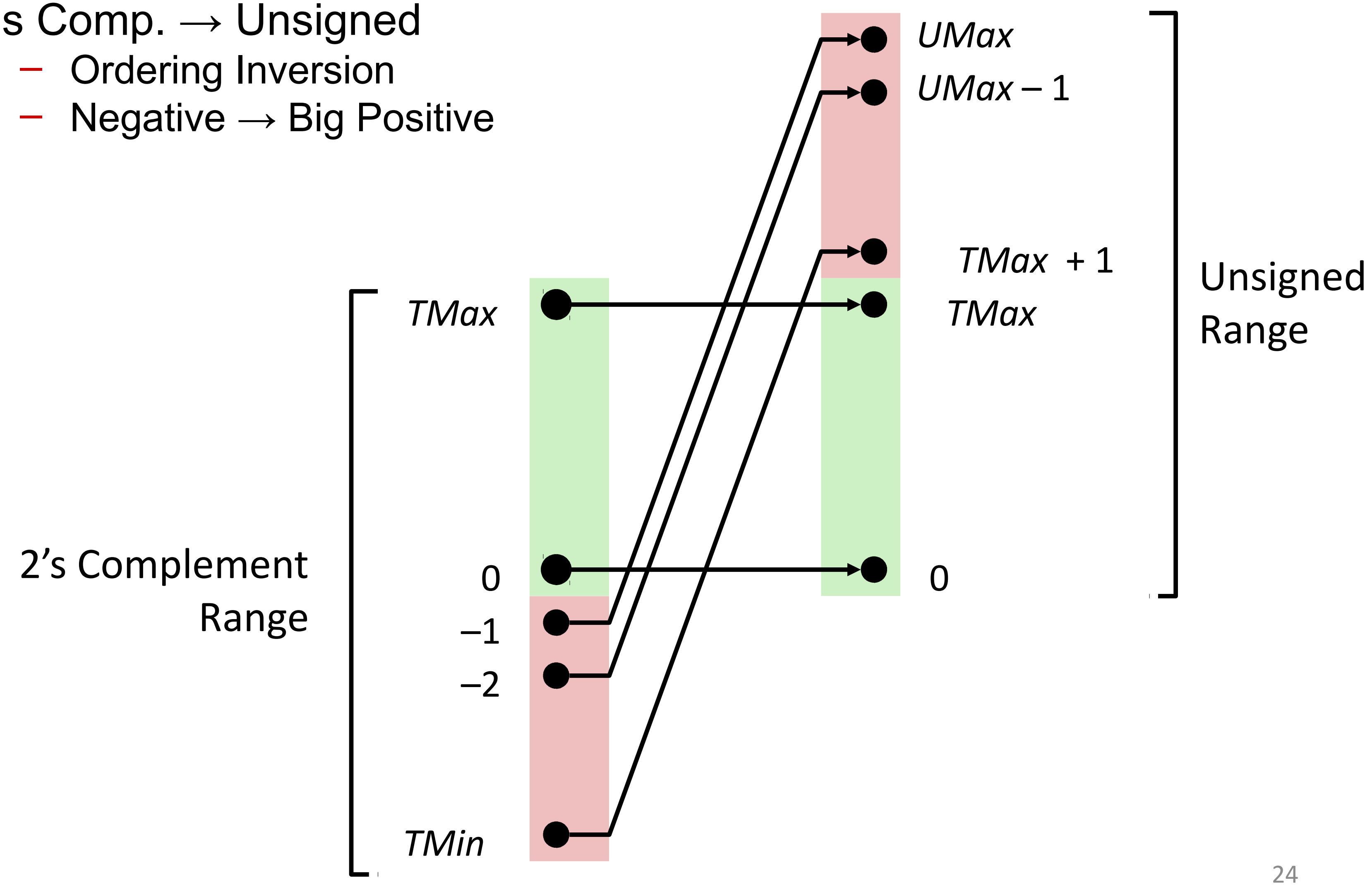


Maintain Same Bit Pattern



Bit-Level Operations in C

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



Proof

Unsigned

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

Two’s Complement

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

$$B2U(X) - B2T(X) = x_{w-1} \cdot 2^w$$

$$B2U(X) = x_{w-1} \cdot 2^w + B2T(X)$$

$$T2U = B2U(T2B) = x_{w-1} \cdot 2^w + B2T(T2B) = x + x_{w-1} \cdot 2^w = \begin{cases} x, & x \geq 0 \\ x + 2^w, & x < 0 \end{cases}$$

$$T2U(x) = \begin{cases} x, & x \geq 0 \\ x + 2^w, & x < 0 \end{cases}$$

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    - `0U, 4294967259U`

- 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;`
    - `uy = ty;`

Signed vs. Unsigned in C

- 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 = -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)	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

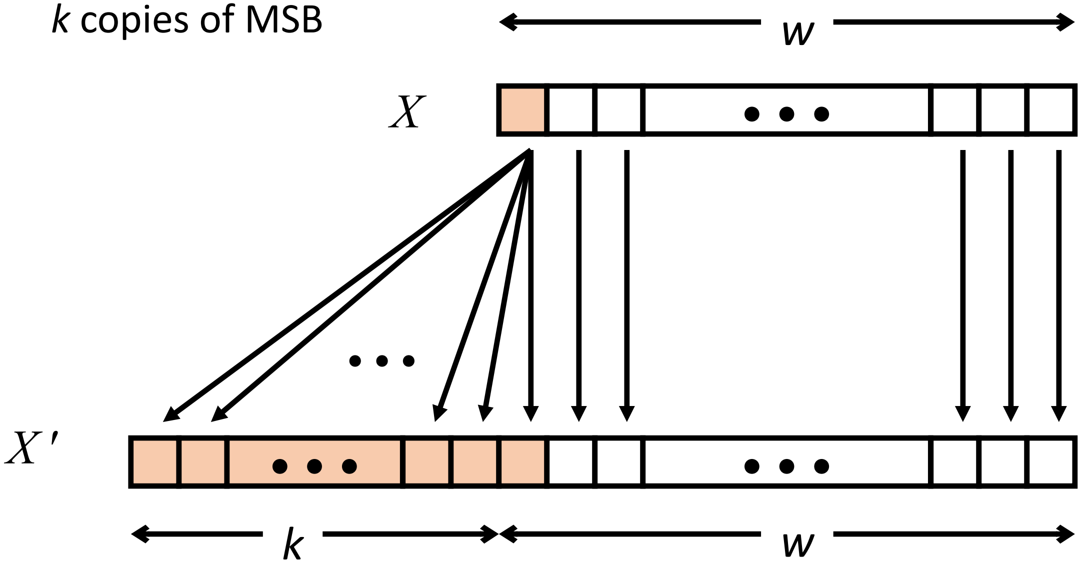
Signed vs. Unsigned in C

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2w

- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

Sign Extension

- Task:
  - Given w-bit signed integer *x*
  - Convert it to w+k-bit integer with same value
- Rule:
  - 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 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

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Summary:
Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

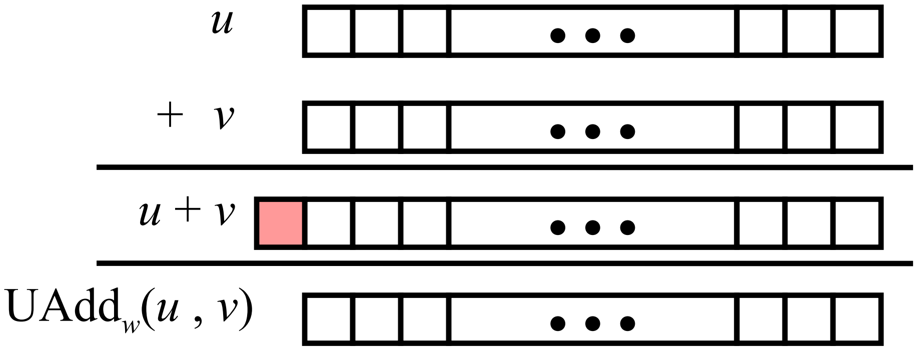
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output

- Implements Modular Arithmetic
  - s = $\mathrm{UAdd}_w(u , v) = u + v \mod 2^w$

Mathematical Properties

- Modular Addition Forms an Abelian Group
  - Closed under addition

$$0 \le \mathrm{UAdd}_w(u\,,\,v) \;\le\; 2^w - 1$$

  - Commutative

$$\mathrm{UAdd}_w(u\,,\,v) \;=\; \mathrm{UAdd}_w(v\,,\,u)$$

  - Associative

$$\mathrm{UAdd}_w(t,\,\mathrm{UAdd}_w(u\,,\,v)) \;=\; \mathrm{UAdd}_w(\mathrm{UAdd}_w(t,\,u\,),\,v)$$

  - 0 is additive identity

$$\mathrm{UAdd}_w(u\,,\,0) \;=\; u$$

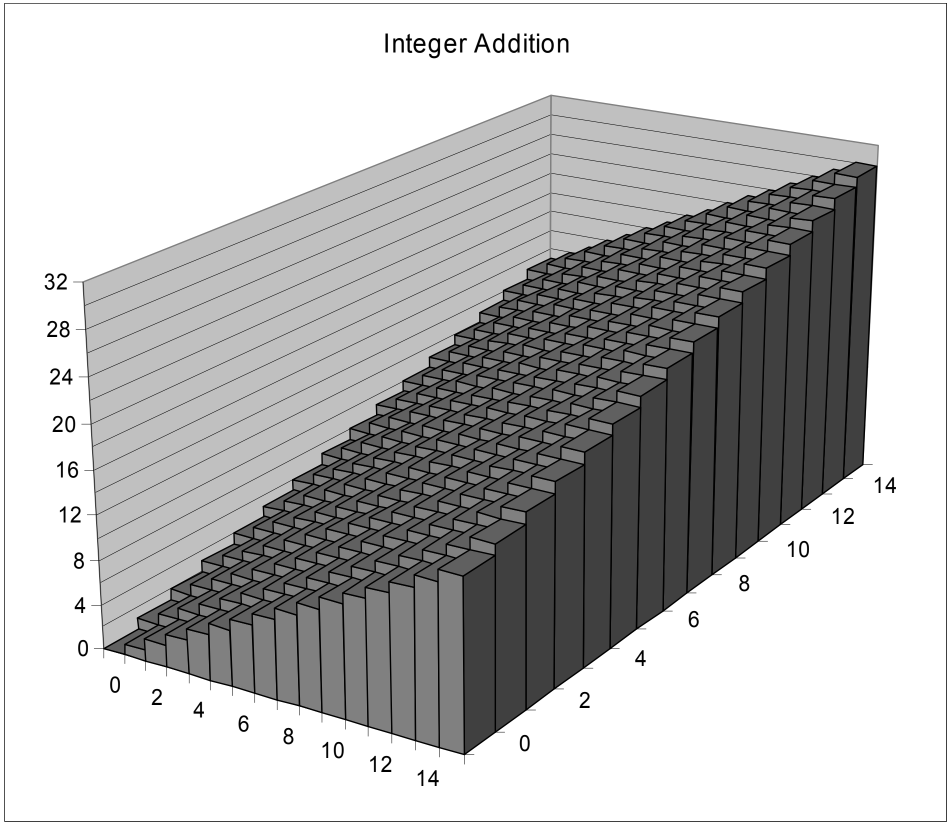
  - Every element has additive inverse

Let $\quad \mathrm{UComp}_w(u\,) \;=2^w - u$

$$\mathrm{UAdd}_w(u\,,\,\mathrm{UComp}_w(u\,)) \;=\; 0$$

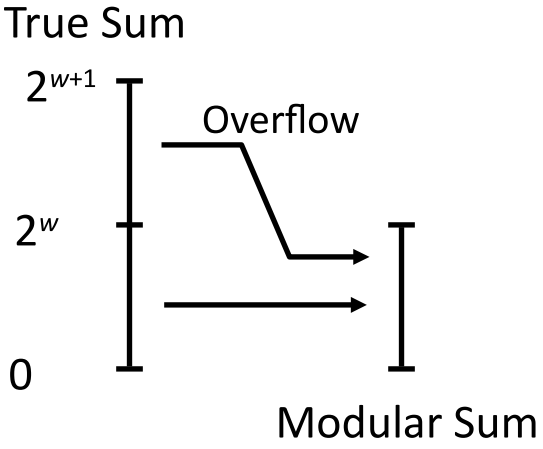
Visualizing (Mathematical) Integer Addition

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



Visualizing Unsigned Addition

- Wraps Around
  - If true sum ≥ 2^w
  - At most once



True Sum

2^{w+1}

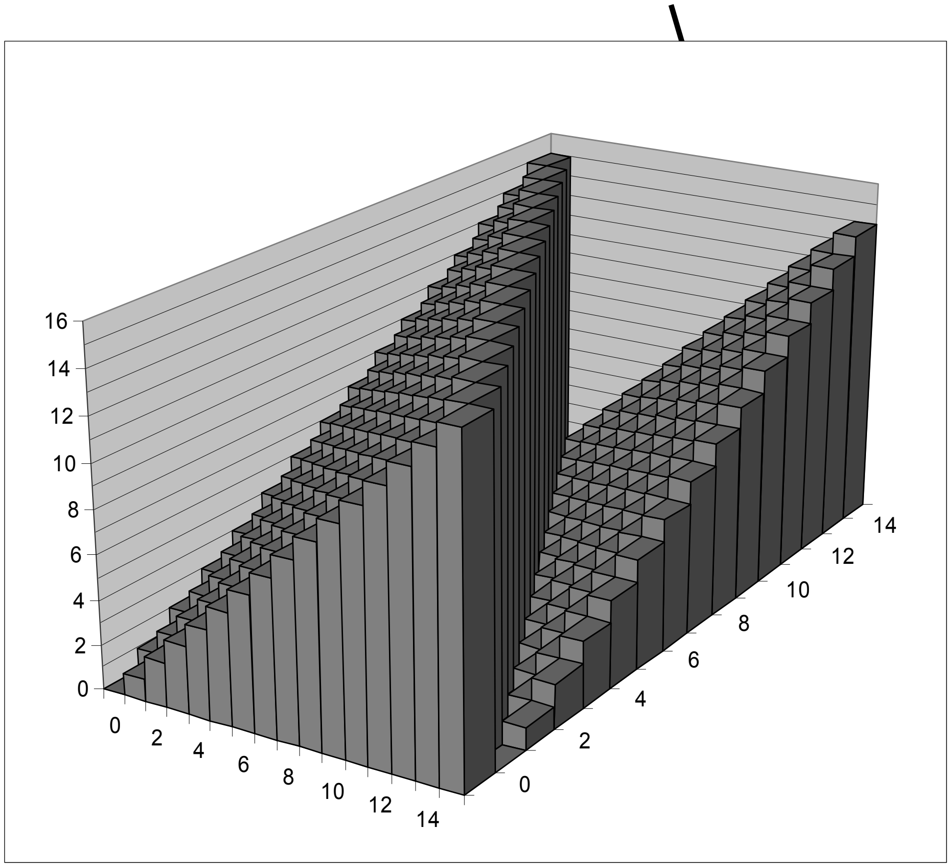
Overflow

2^w

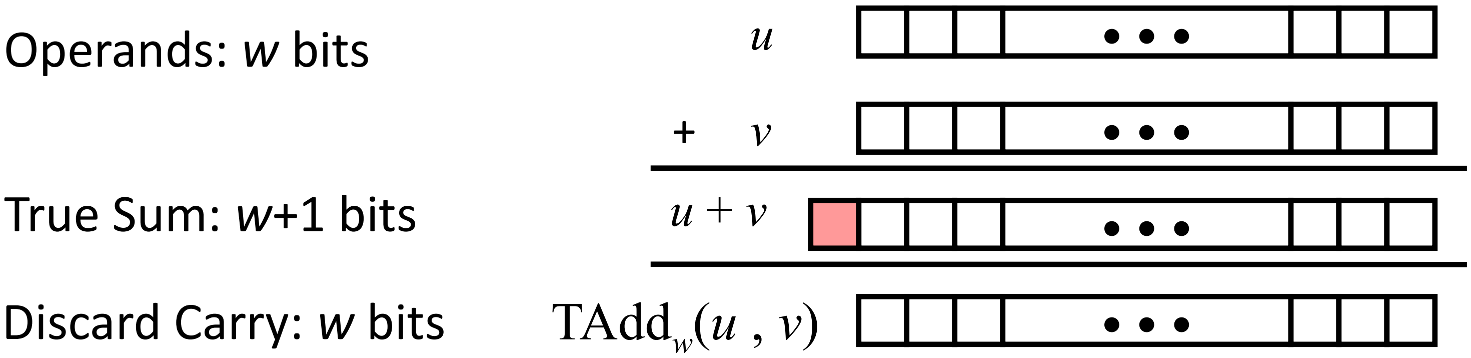
0

Modular Sum

Overflow



Two’s Complement Addition



Operands: *w* bits

u

$+ \quad v$

True Sum: *w*+1 bits

$u + v$

Discard Carry: *w* bits

$\mathrm{TAdd}_w(u\ ,\ v)$

- 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`

Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  - $\text{TAdd}_w(u\ ,\ v) = \ \text{U2T}(\text{UAdd}_w(\text{T2U}(u\ ), \text{T2U}(v)))$
    - Since both have identical bit patterns

- Two's Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

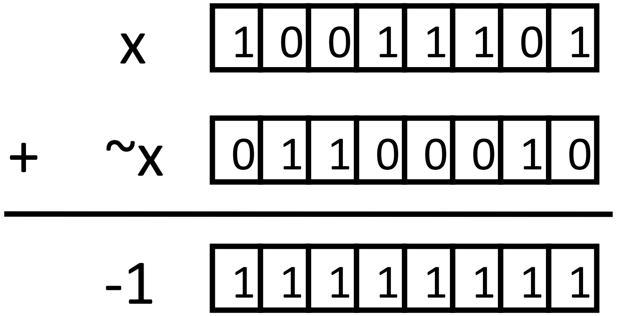
$$TComp_w(u) \quad = \quad \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Mathematical Properties of TAdd

- Claim: Following Holds for 2’s Complement

  `~x + 1 == -x`

- Complement
  - Observation: `~x + x == 1111…111 == -1`

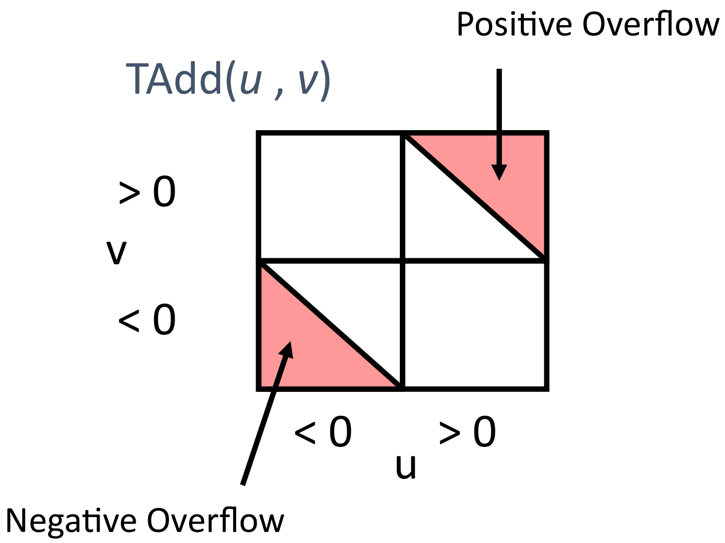


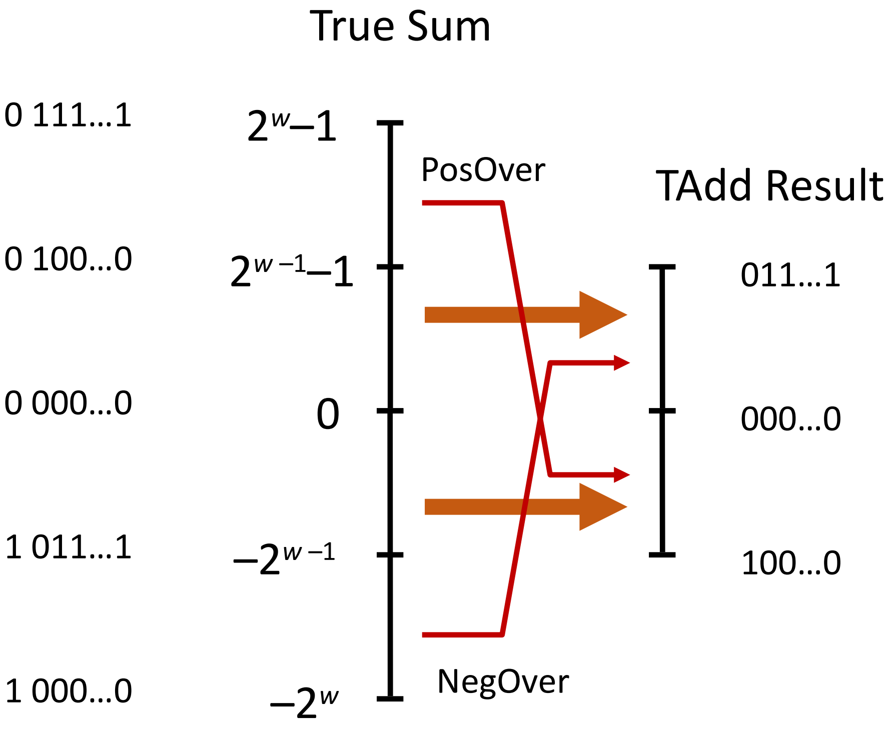
	x	1	0	0	1	1	1	0	1
+	~x	0	1	1	0	0	0	1	0
	-1	1	1	1	1	1	1	1	1

- Complete Proof?

TAdd Overflow

- Functionality
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer



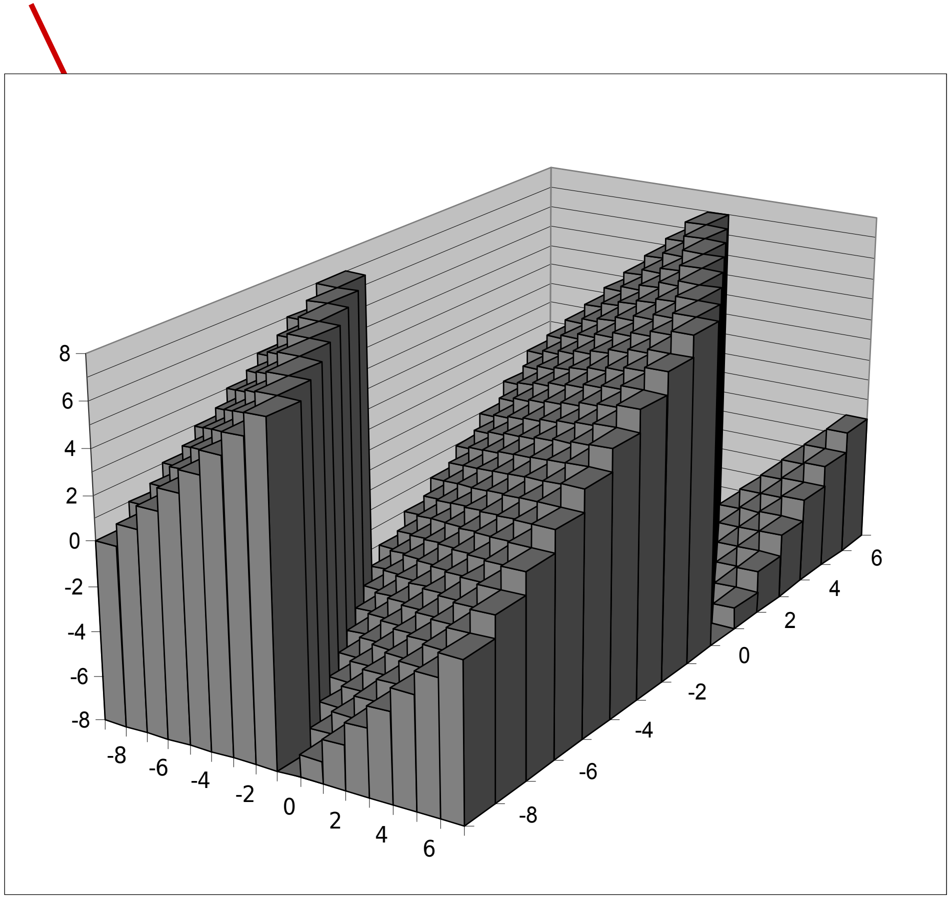


$$TAdd_w(u,v) \;=\; \begin{cases} u+v+2^{w-1} & u+v < TMin_w \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \end{cases}$$

Visualizing 2's Complement Addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum ≥ 2^{w-1}
    - Becomes negative
    - At most once
  - If sum < -2^{w-1}
    - Becomes positive
    - At most once

NegOver



Multiplication

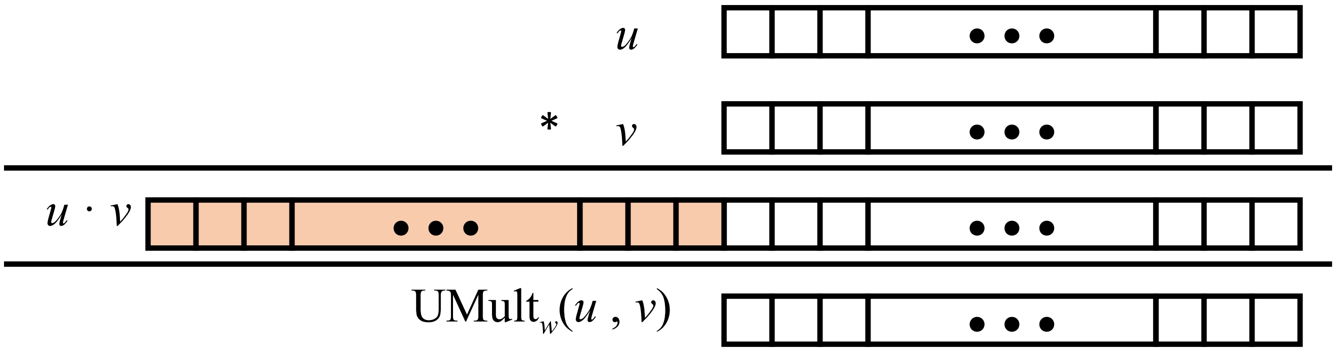
- Goal: Computing Product of *w*-bit numbers *x, y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed 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



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\mathrm{UMult}_w(u, v) = u \cdot v \mod 2^w$

Properties of Unsigned Arithmetic

- Unsigned Multiplication with Addition Forms Commutative Ring
  - Addition is commutative group
  - Closed under multiplication

$$0 \leq \mathrm{UMult}_w(u\ ,\ v) \ \leq\ 2^w - 1$$

  - Multiplication Commutative

$$\mathrm{UMult}_w(u\ ,\ v) \ =\ \mathrm{UMult}_w(v\ ,\ u)$$

  - Multiplication is Associative

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

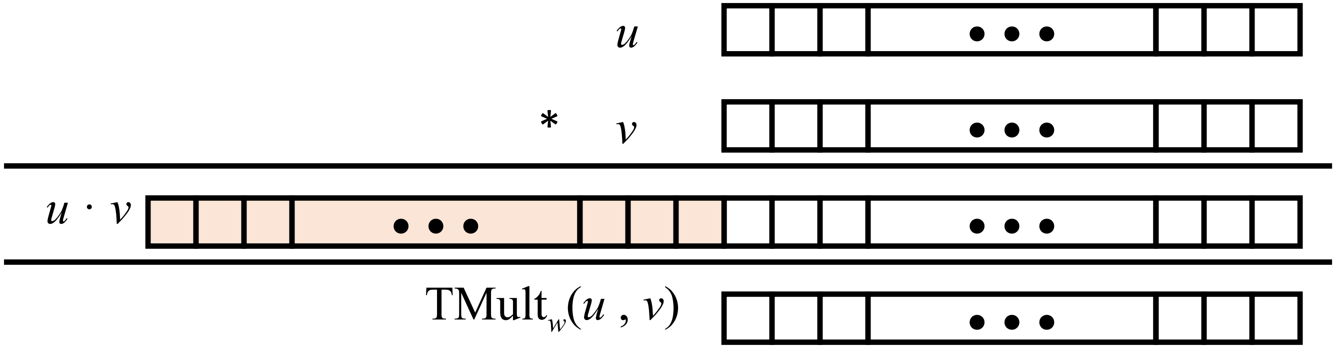
  - 1 is multiplicative identity

$$\mathrm{UMult}_w(u\ ,\ 1) \ =\ u$$

  - Multiplication distributes over addtion

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

Signed Multiplication in C



Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits

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

Properties of Two's Comp. Arithmetic

- Isomorphic Algebras
  - Unsigned multiplication and addition
    - Truncating to w bits
  - Two's complement multiplication and addition
    - Truncating to w bits

- Both Form Rings
  - Isomorphic to ring of integers mod 2^w

- Comparison to (Mathematical) Integer Arithmetic
  - Both are rings
  - Integers obey ordering properties, e.g.,

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

$$u > 0,\, v > 0 \quad \Rightarrow \quad u \cdot v > 0$$

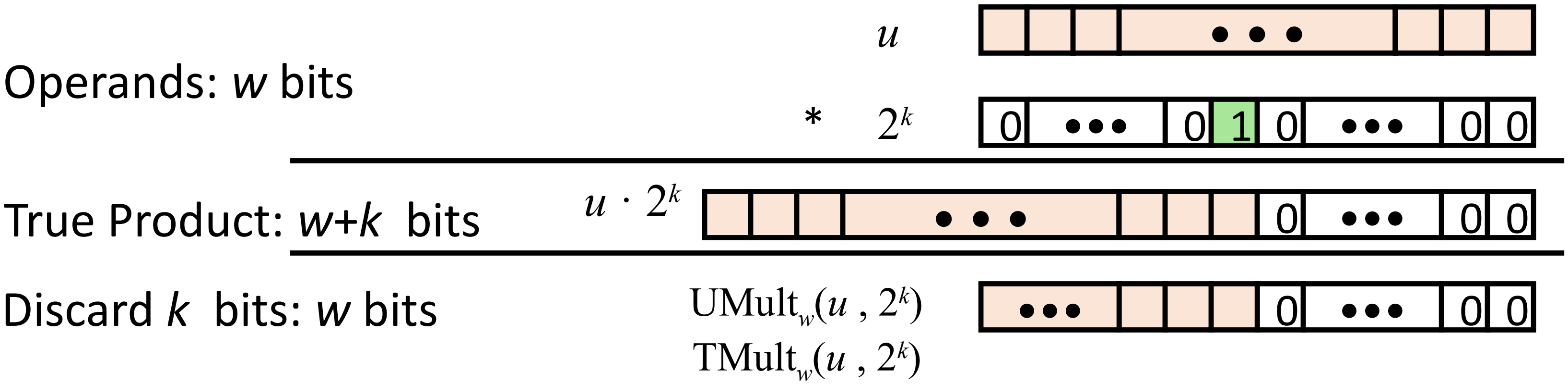
  - These properties are not obeyed by two's comp. arithmetic

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

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

Power-of-2 Multiply with Shift

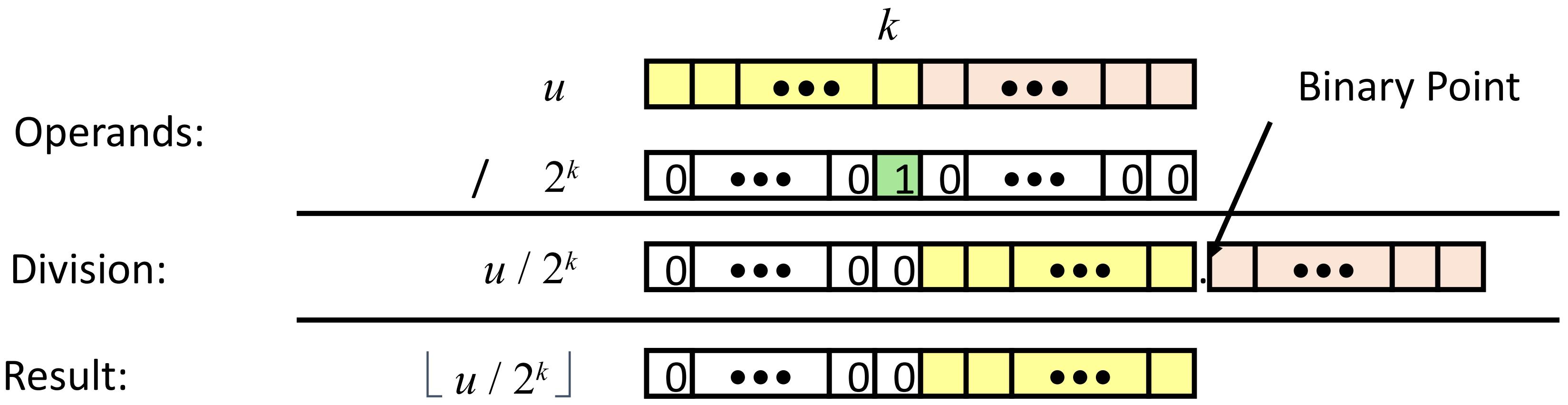
- Operation
  - u << k gives u * 2^k
  - Both signed and unsigned



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

Unsigned Power-of-2 Divide with Shift

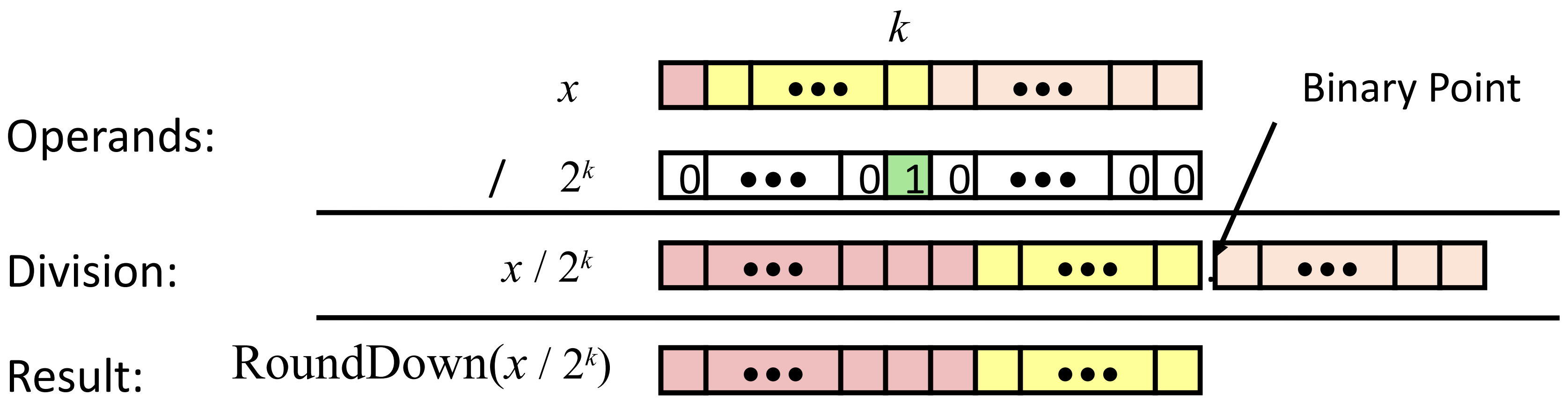
- Quotient of Unsigned by Power of 2
- **`u >> k`** gives ⌊ **`u / 2k`** ⌋
- Uses logical shift



	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`

Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - `x >> k` gives $\lfloor$ `x /` $2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when u < 0



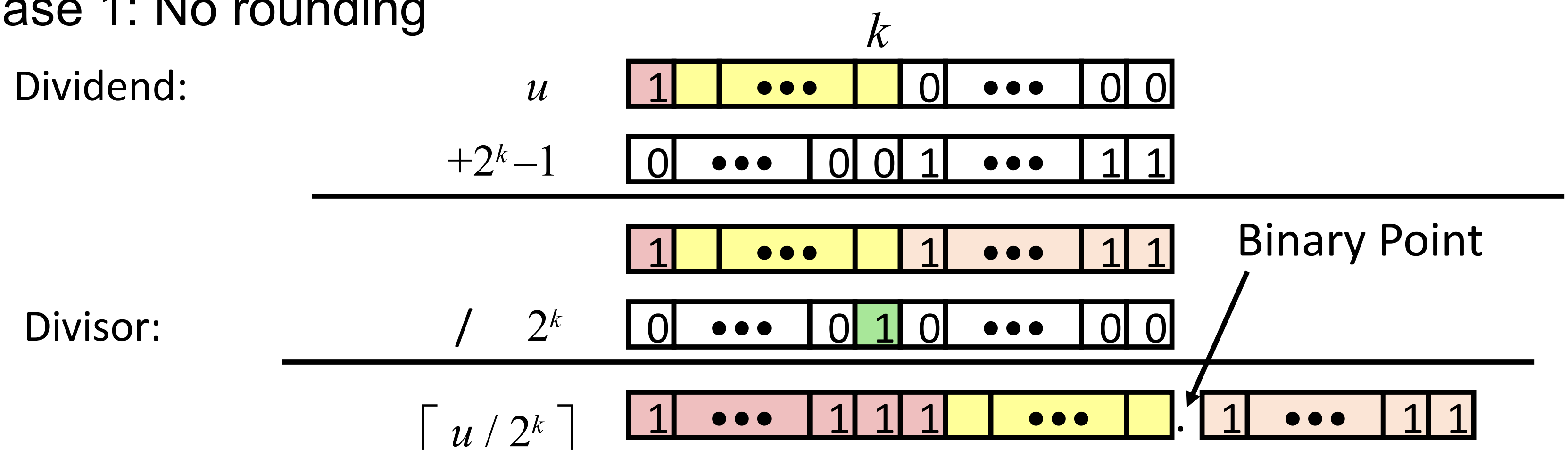
	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want $\lceil$ **x / 2^k** $\rceil$ (Round Toward 0)
  - Compute as $\lfloor$ **(x+2^k-1)/ 2^k** $\rfloor$

  - In C: **`(x + (1<<k)-1) >> k`**
  - Biases dividend toward 0

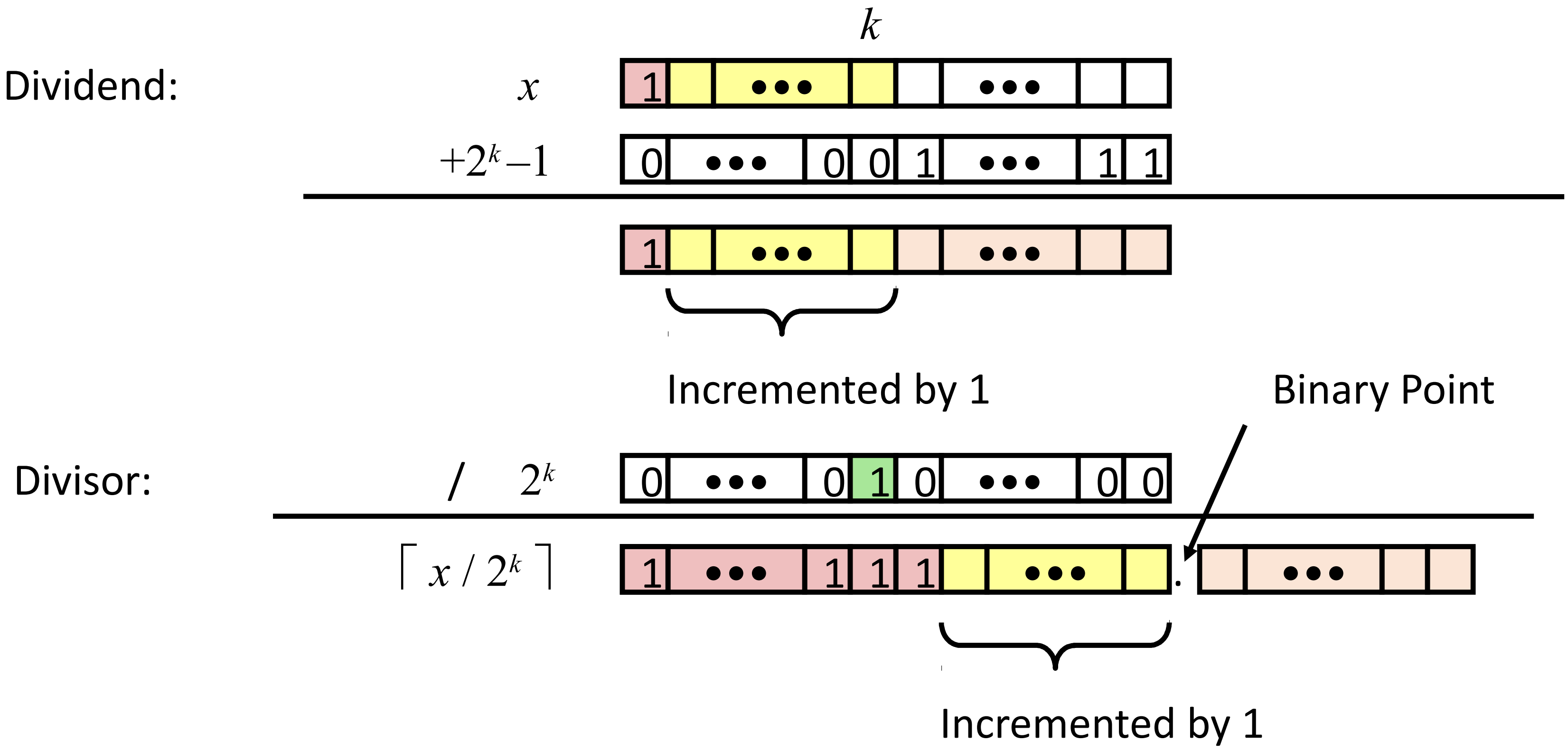
- Case 1: No rounding



Biasing has no effect

Mathematical Properties

- Case 2: Rounding



Biasing adds 1 to final result

Arithmetic: Basic Rules

- Addition:
  - 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

- Multiplication:
  - 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

- Unsigned ints, 2’s complement ints are isomorphic rings: isomorphism = casting

- Left shift
  - Unsigned/signed: multiplication by 2^k
  - Always logical shift

- Right shift
  - 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

Why Should I Use Unsigned?

- Don’t use without understanding implications
  - 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

- Proper way to use unsigned as loop index

```
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*

- Even better

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

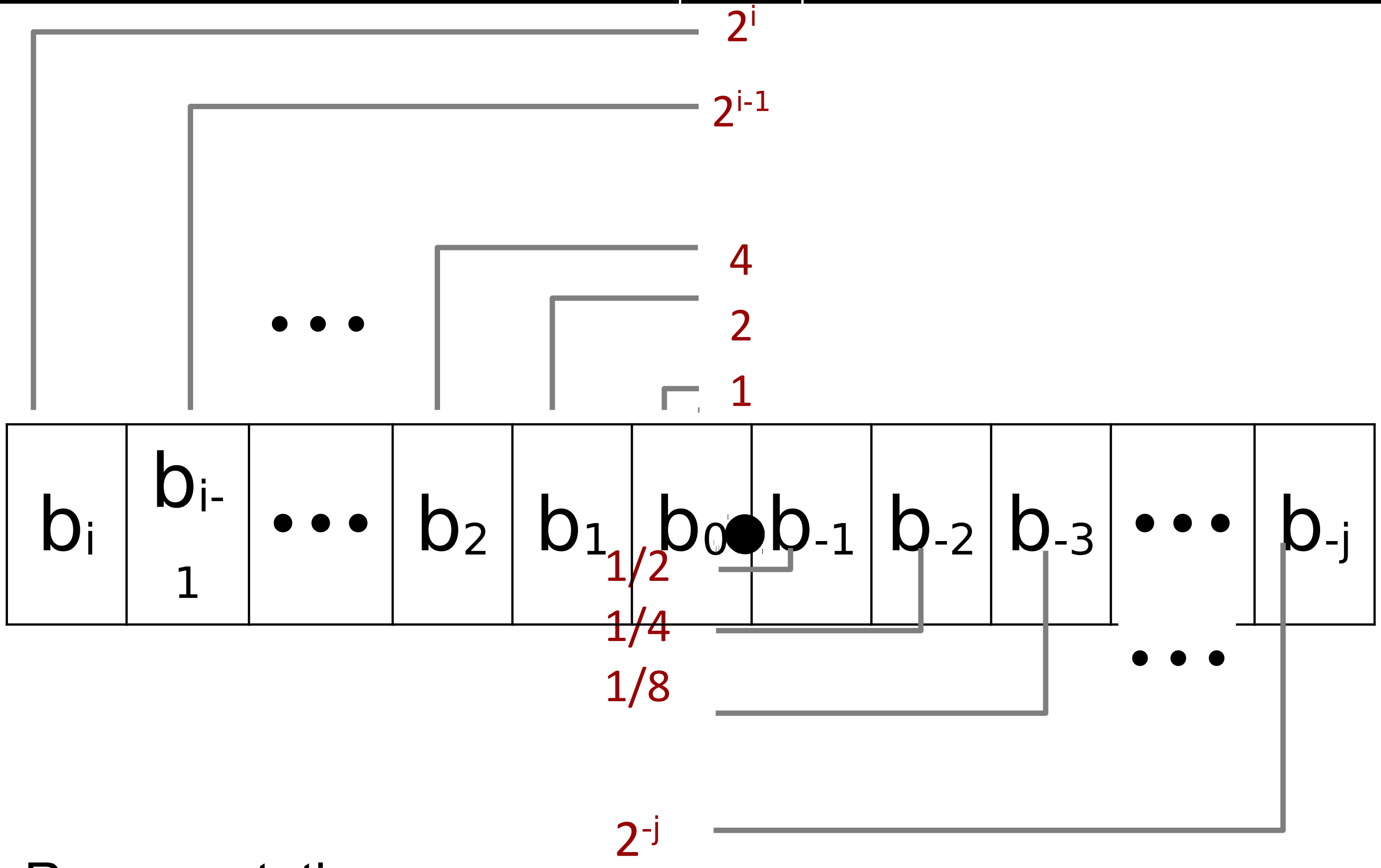
  - Data type size_t defined as unsigned value with length = word size
  - Code will work even if cnt = *UMax*
  - What if **cnt** is signed and < 0?

Why Should I Use Unsigned? (cont.)

- Do Use When Performing Modular Arithmetic
  - Multiprecision arithmetic

- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Fractional Binary Numbers: Examples

- Value Representation

5 3/4	101.112
2 7/8	010.1112
1 7/16	001.01112

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - 1/2 + 1/4 + 1/8 + … + 1/2i + … → 1.0
    - Use notation 1.0 – ε

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form x/2k
    - Other rational 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
  - Just one setting of binary point within the w bits
  - 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

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

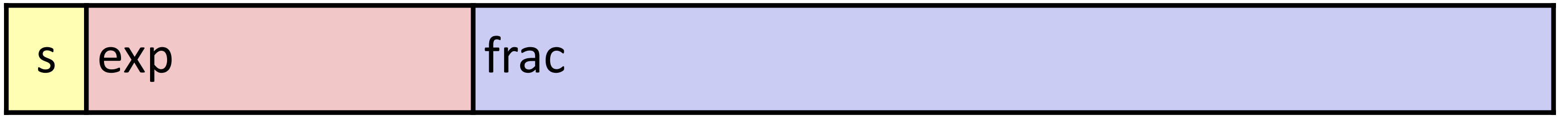
Floating Point Representation

- Numerical Form:

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

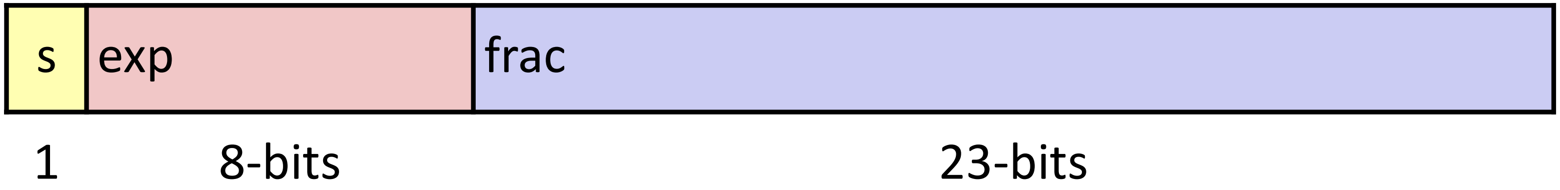
  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

- Encoding
  - MSB ***s*** is sign bit s
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)

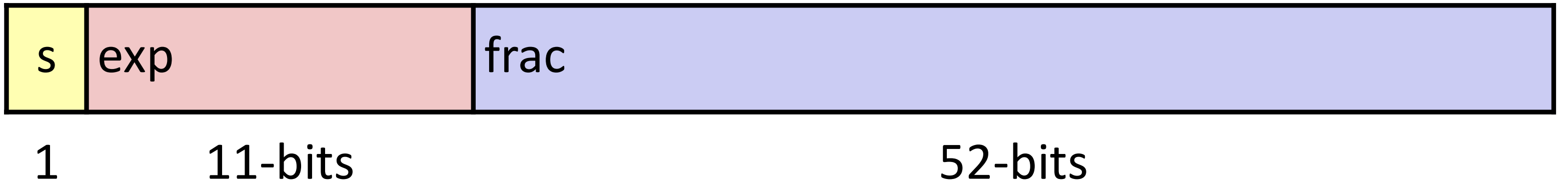


Precision options

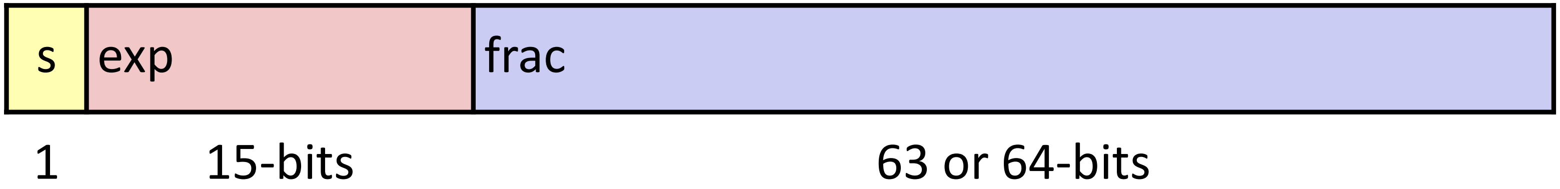
- Single precision: 32 bits



- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



“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

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

- Value: `float F = 15213.0;`

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \times 2^{13}$

- Significand

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

  $\texttt{frac} = \underline{\texttt{1101101101101}}\texttt{0000000000}_2$

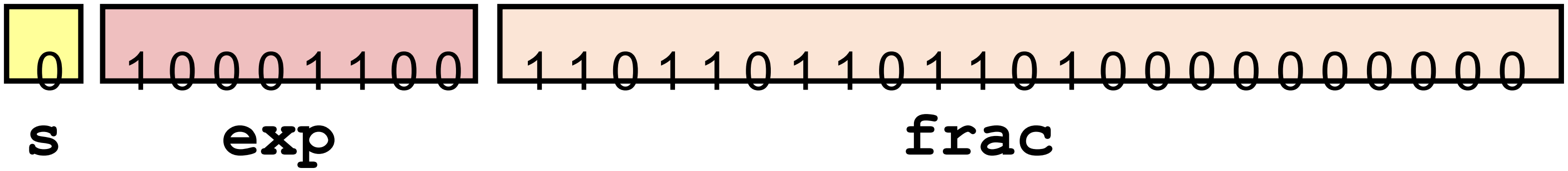
- Exponent

  $E = 13$

  $Bias = 127$

  $Exp = 140 = \texttt{10001100}_2$

- Result:



0 | 1 0 0 0 1 1 0 0 | 1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0

Denormalized Values

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

$E = 1 - \text{Bias}$

- Condition: exp = 000…0
- Exponent value: E = 1 – 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
    - Note distinct values: +0 and –0 (why?)
  - exp = 000…0, frac ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

Examples

code1.c

```
#include <iostream>
#include <string>
using name
int main(
    const
    const
    float
    for(i
    {
        y
        y
        y+=0.1f;
        y-=0.1f;
    }
    return 0;
}
```

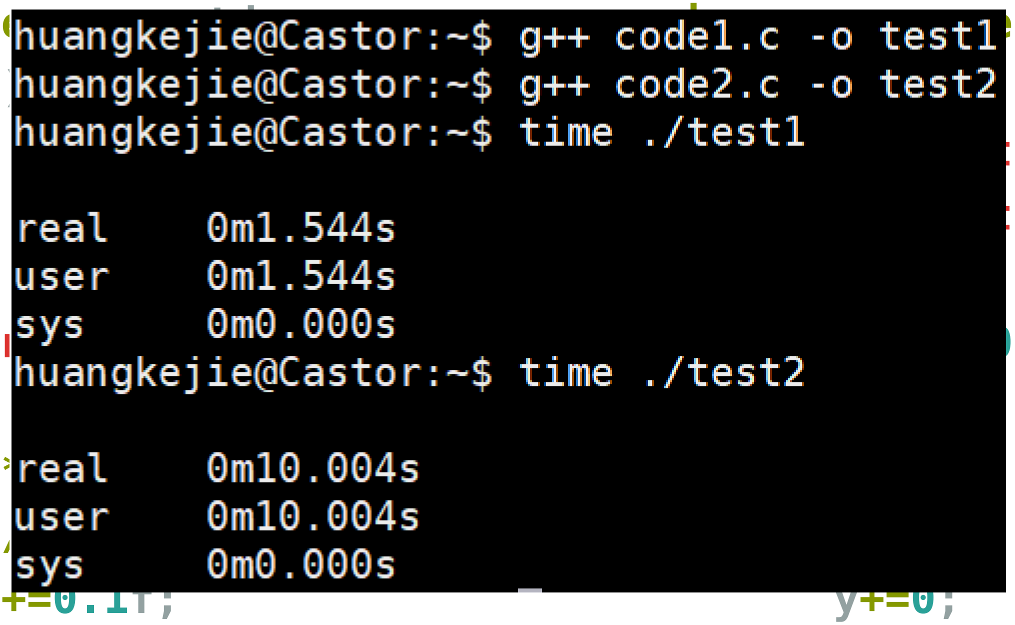
code2.c

```
#include <iostream>
#include <string>
                     std;

                   x=1.1;
                   z=1.123;

               ;j<90000000;j++)


        y+=0;
        y-=0;
    }
    return 0;
}
```



```
huangkejie@Castor:~$ g++ code1.c -o test1
huangkejie@Castor:~$ g++ code2.c -o test2
huangkejie@Castor:~$ time ./test1

real    0m1.544s
user    0m1.544s
sys     0m0.000s
huangkejie@Castor:~$ time ./test2

real    0m10.004s
user    0m10.004s
sys     0m0.000s
```

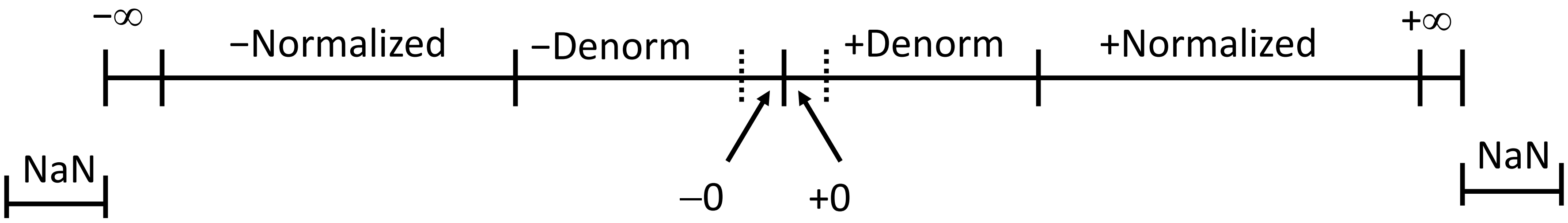
Special Values

- Condition: exp = **`111…1`**

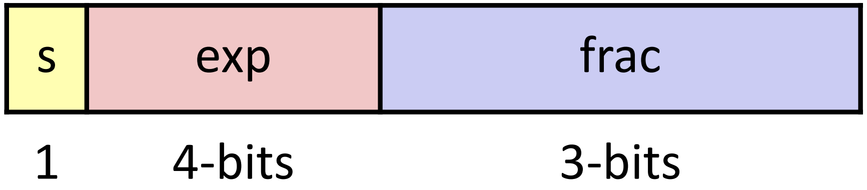
- Case: exp = **`111…1`**, frac = **`000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = + ∞, 1.0/−0.0 = − ∞

- Case: exp = **`111…1`**, frac ≠ **`000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞ , ∞ × 0

Visualization: Floating Point Encodings



Tiny Floating Point Example



- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the frac

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

Visualization: Floating Point Encodings

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

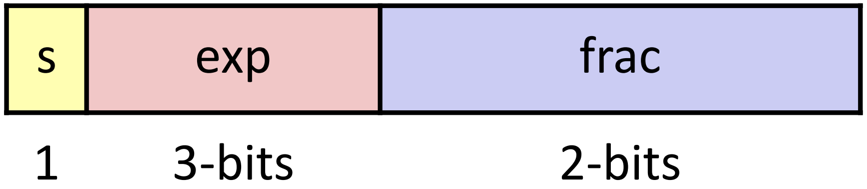
n: $E = \text{Exp} - \text{Bias}$

d: $E = 1 - \text{Bias}$

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	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
	0	0111	000	0	8/8*1 = 1	
	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	

Distribution of Values

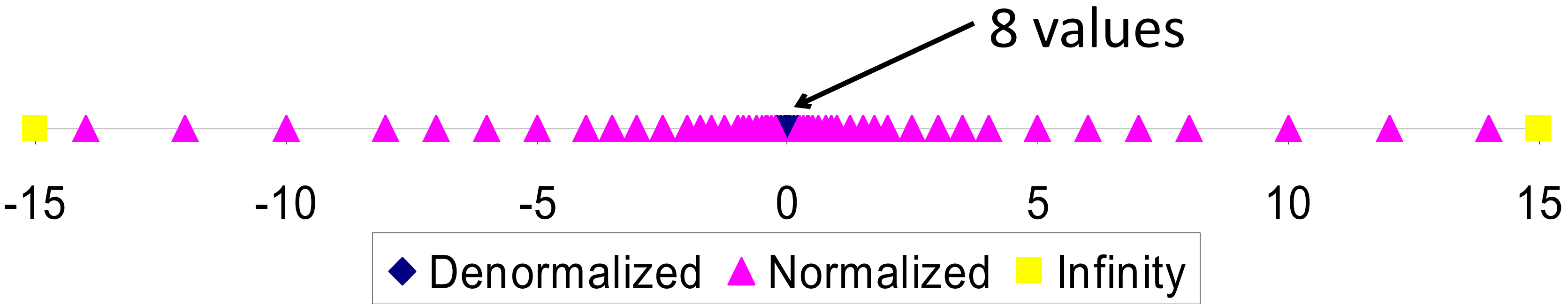
- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



s | exp | frac

1 | 3-bits | 2-bits

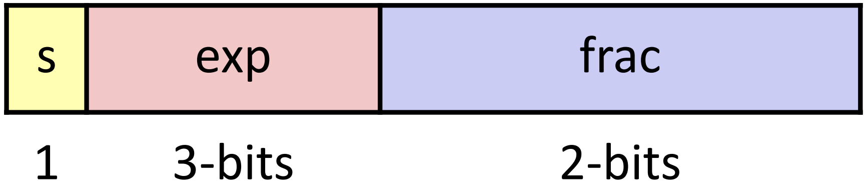
- Notice how the distribution gets denser toward zero.



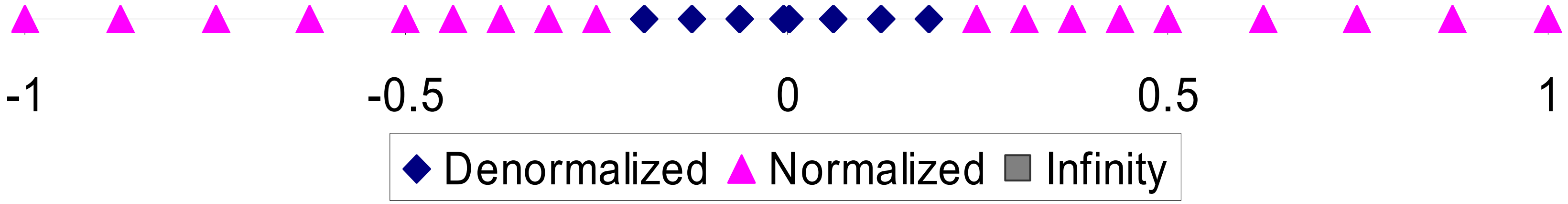
8 values

Distribution of Values (close-up view)

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


s | exp | frac

1 | 3-bits | 2-bits



Special Properties of the IEEE Encoding

- FP Zero Same as Integer Zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

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

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

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

Rounding

- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards 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

Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
- Sum of set 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)
7.8850000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers
  - “Even” when least significant bit is 0
  - “Half way” when bits to right of rounding position = $100\ldots_2$

- Examples
  - 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 ¼
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

FP Multiplication

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

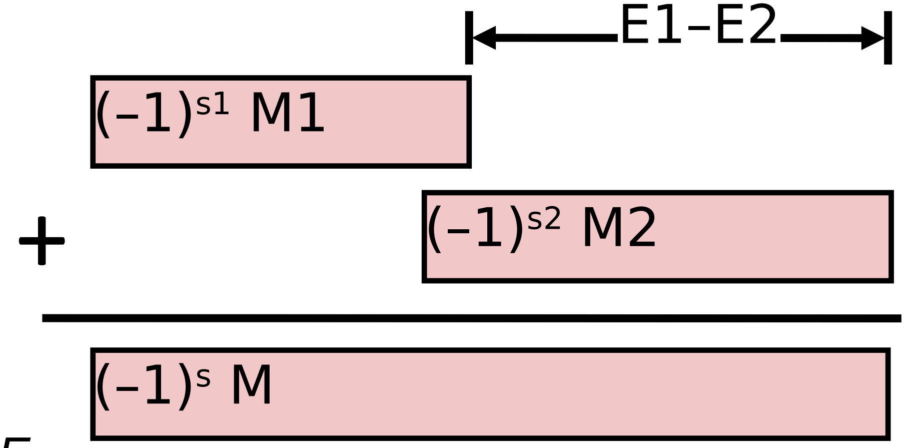
- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If E out of range, overflow
  - Round *M* to fit frac precision

- 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*

Get binary points lined up



- Fixing
  - If *M* ≥ 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

- Compare to those of 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
    - Yes, except for infinities & NaNs
- Monotonicity
  - a ≥ b ⇒ a+c ≥ b+c? Almost
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- Compare to 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
    - 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`
- Monotonicity
  - $a \geq b \;\; \& \; c \geq 0 \;\; \Rightarrow a * c \geq b * c$? Almost
    - Except for infinities & NaNs

Floating Point in C

- C Guarantees Two Levels

  `float` single precision
  float double precision

- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double/float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

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 = …;
```

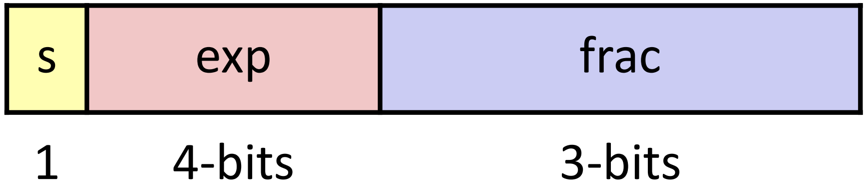
Assume neither d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(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`

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
- As if computed with perfect precision and then rounded
- Not the same as real arithmetic
- Violates associativity/distributivity
- Makes life difficult for compilers & serious numerical applications programmers

Creating Floating Point Number



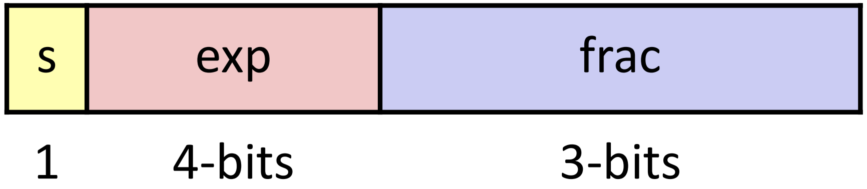
- Steps
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

- Case Study
  - 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



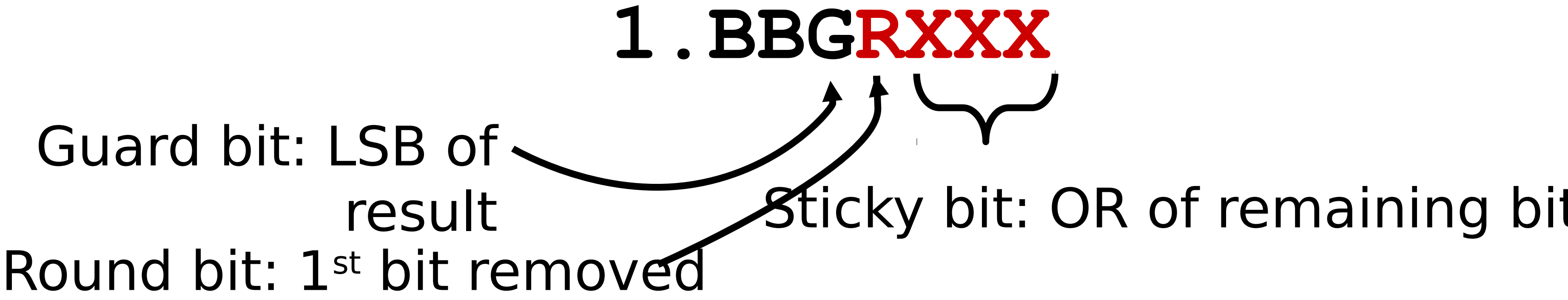
s exp frac

1 4-bits 3-bits

- Requirement
  - 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

Rounding



1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bit

- 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

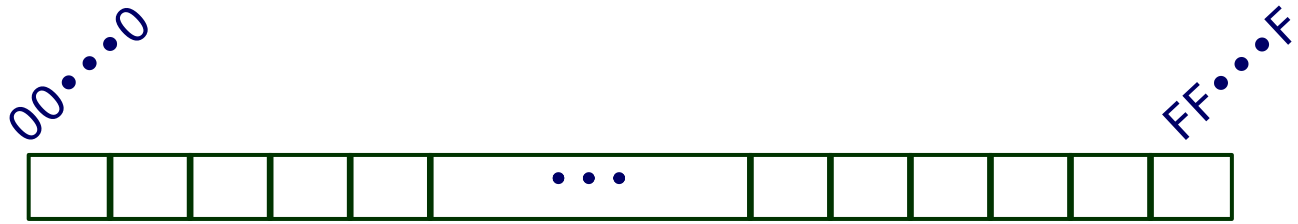
Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Interesting Numbers

{single,double}

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

Byte-Oriented Memory Organization



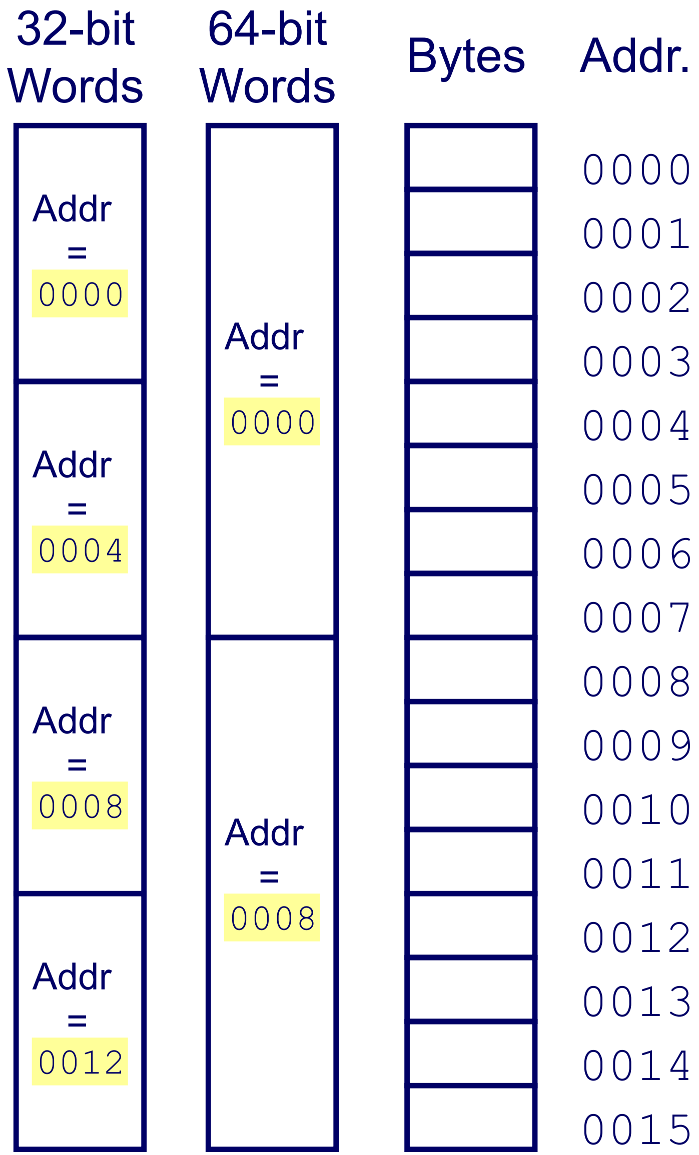
- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- Note: system provides private address spaces to each "process"
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (232 bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That’s 18.4 X 10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization



- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

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
`long double`	–	–	10/16
pointer	4	8	8

Byte Ordering

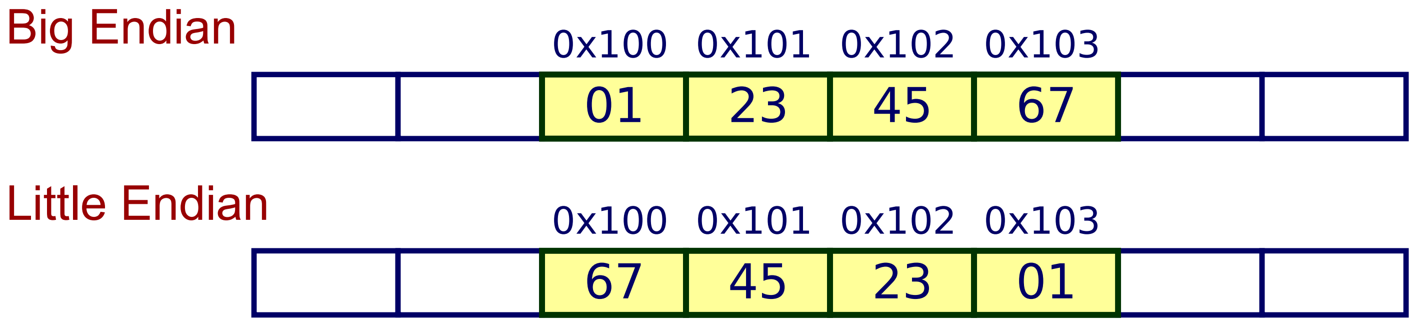
- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address

  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Byte Ordering Example

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



	0x100	0x101	0x102	0x103
Big Endian	01	23	45	67
Little Endian	67	45	23	01

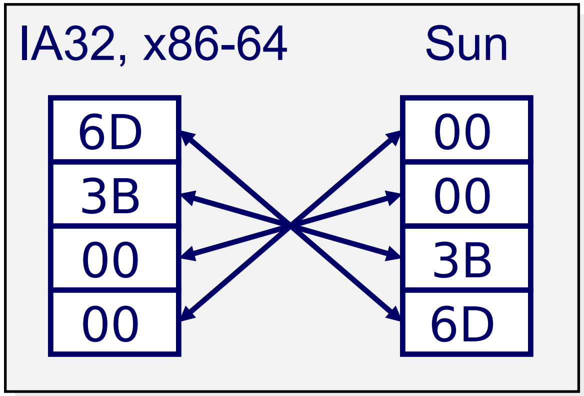
Representing Integers

Decimal: 15213

Binary: 0011 1011 0110 1101

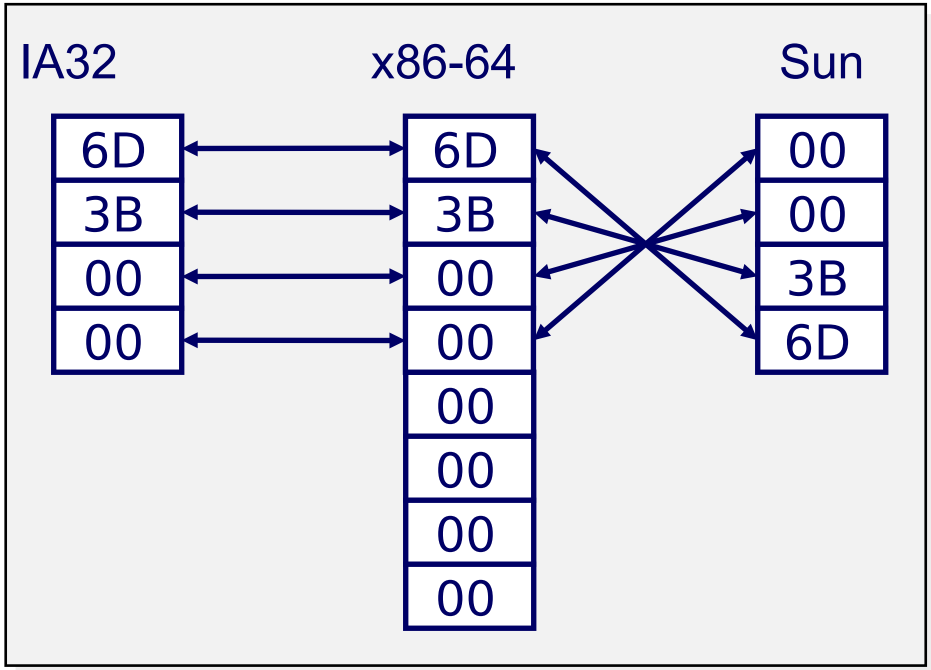
Hex: 3 B 6 D

```
int A = 15213;
```



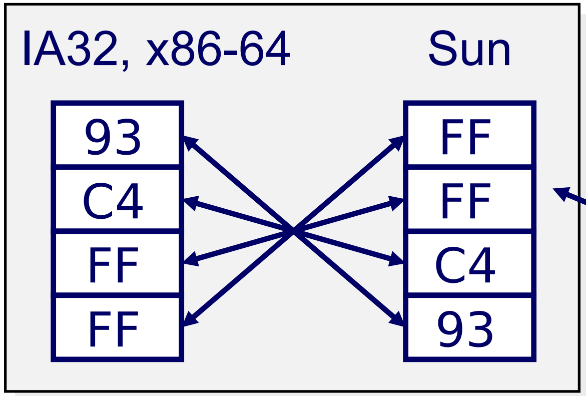
IA32, x86-64	Sun
6D	00
3B	00
00	3B
00	6D

```
long int C = 15213;
```



IA32	x86-64	Sun
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

```
int B = -15213;
```



IA32, x86-64	Sun
93	FF
C4	FF
FF	C4
FF	93

Two’s complement representation

Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void 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");
}
```

Printf directives:

%p:	Print pointer
%x:	Print Hexadecimal

show_bytes Execution Example

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

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

Representing Pointers

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

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

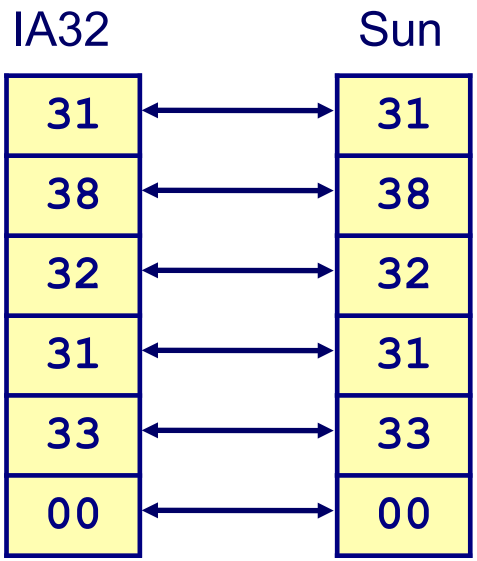
Different compilers & machines assign different locations to objects

Even get different results each time run program

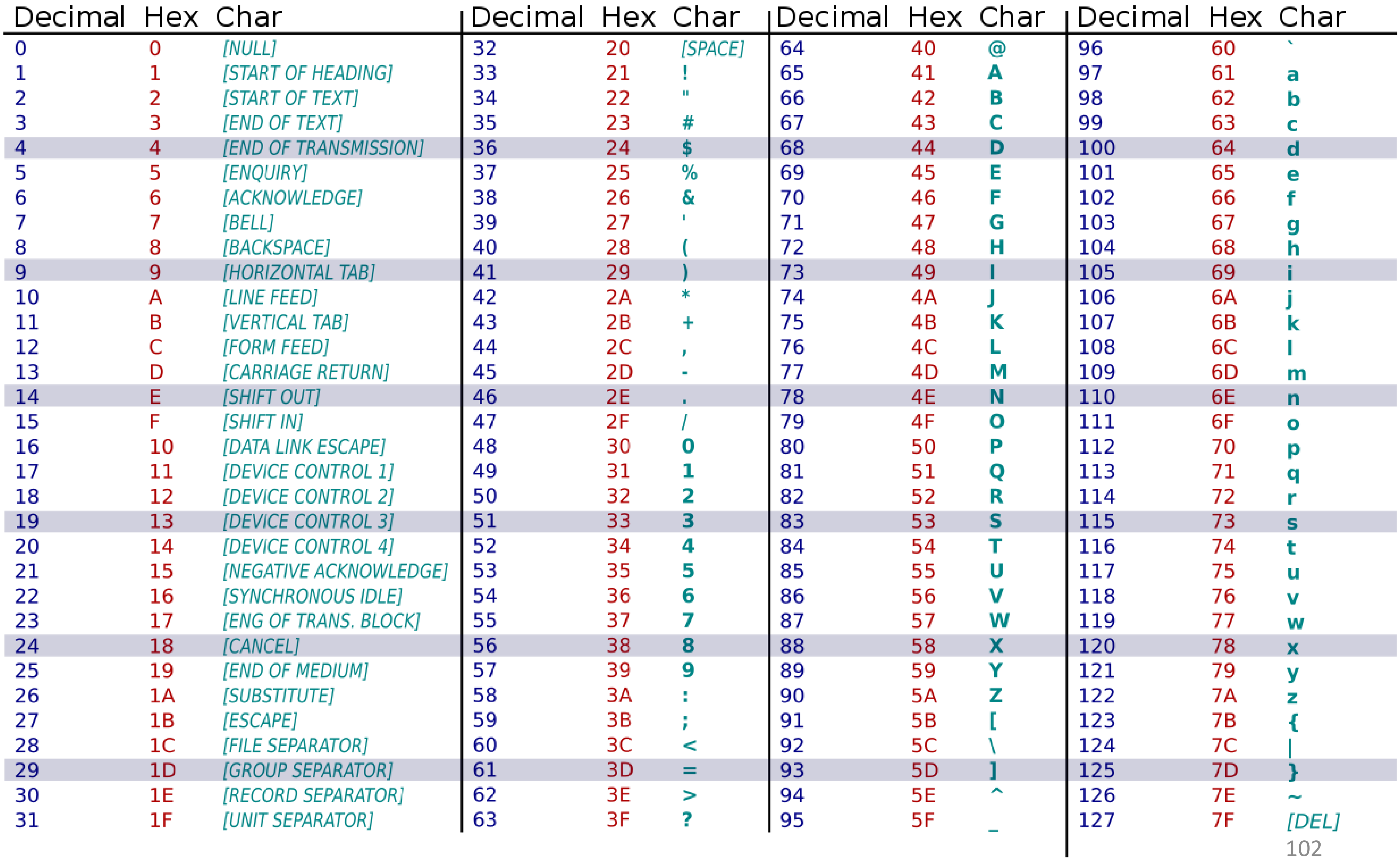
Representing Strings

- Strings in C
  - 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 = 0
- Compatibility
  - Byte ordering not an issue

```
char S[6] = "18213";
```



ASCII Table



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
  leaq (%rax,%rax,2), %rax
  salq $2, %rax
```

Explanation

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

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

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

```
  shrq $3, %rax
```

Explanation

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

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

Compiled Signed Division Code

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

Compiled Arithmetic Operations

```
  testq %rax, %rax
  js    L4
L3:
  sarq $3, %rax
  ret
L4:
  addq $7, %rax
  jmp   L3
```

Explanation

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

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>