Data Types & Operations

'22H2

송 인 식

Outline

- Integers
- Floating points

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**

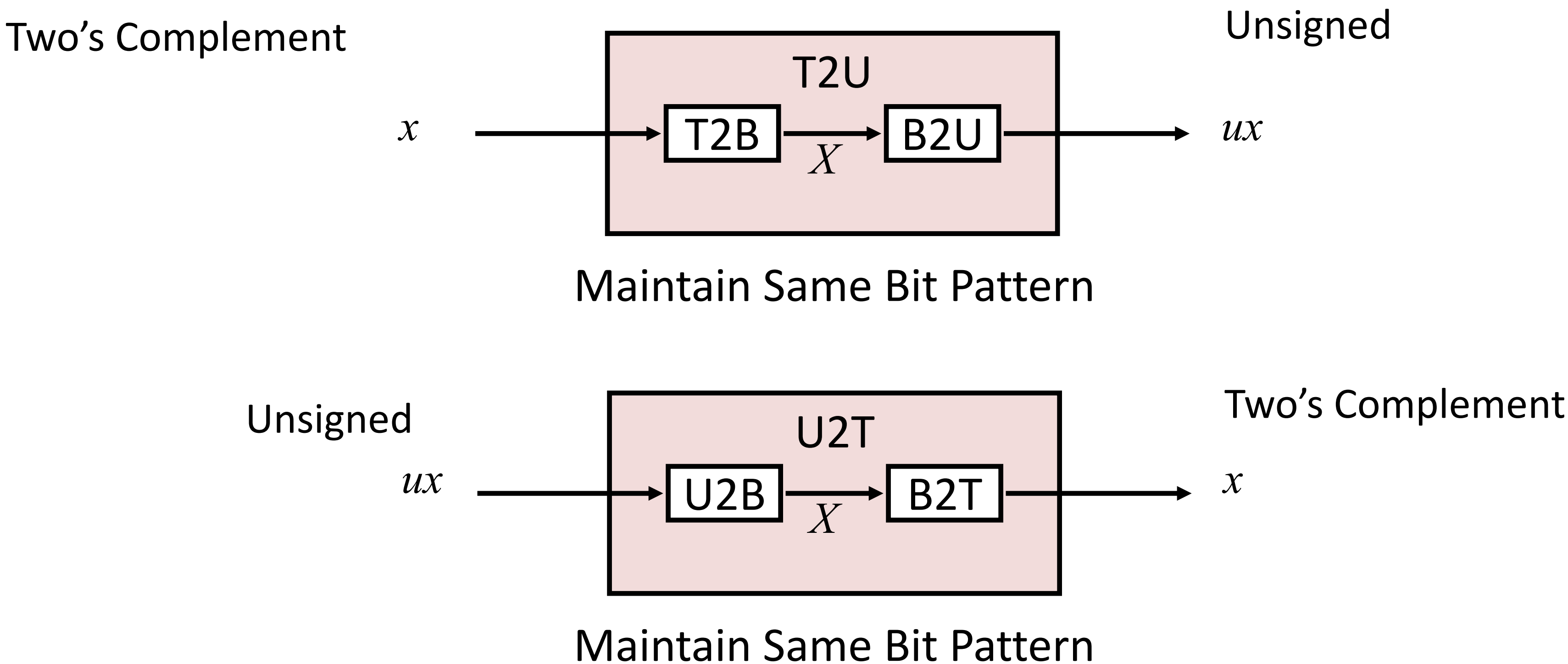
- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Two-complement: Simple Example

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

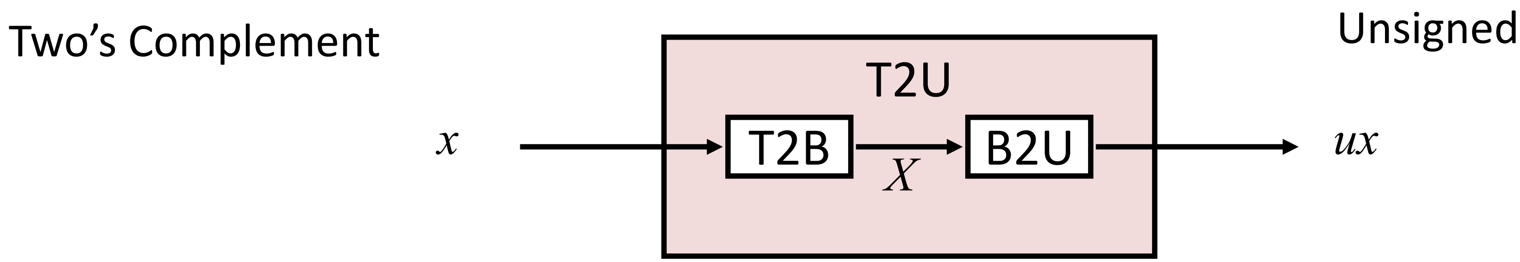
	-16	8	4	2	1	
-10 =	1	0	1	1	0	-16+4+2 = -10

Mapping Between Signed & Unsigned

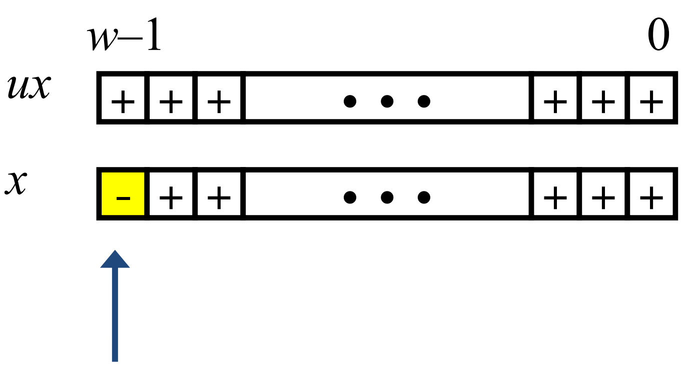


- Mappings between unsigned and two's complement numbers:
  Keep bit representations and reinterpret

Relation between Signed & Unsigned



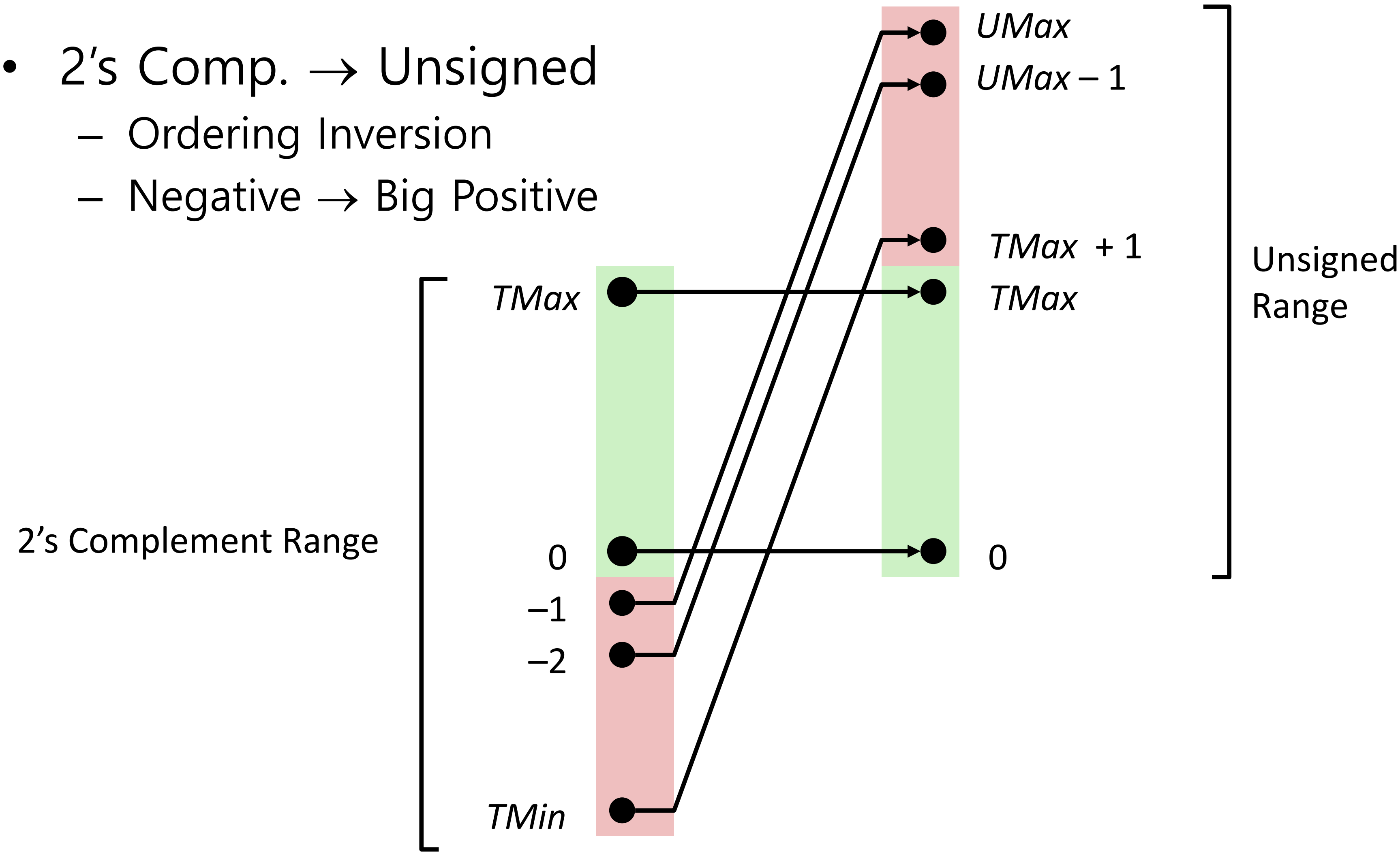
Maintain Same Bit Pattern



Large negative weight
becomes
Large positive weight

Conversion Visualized

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



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;

Casting Surprises

- 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)-1	-2	>	**unsigned**
2147483647	2147483648U	<	**unsigned**
2147483647	(int) 2147483648U	>	**signed**

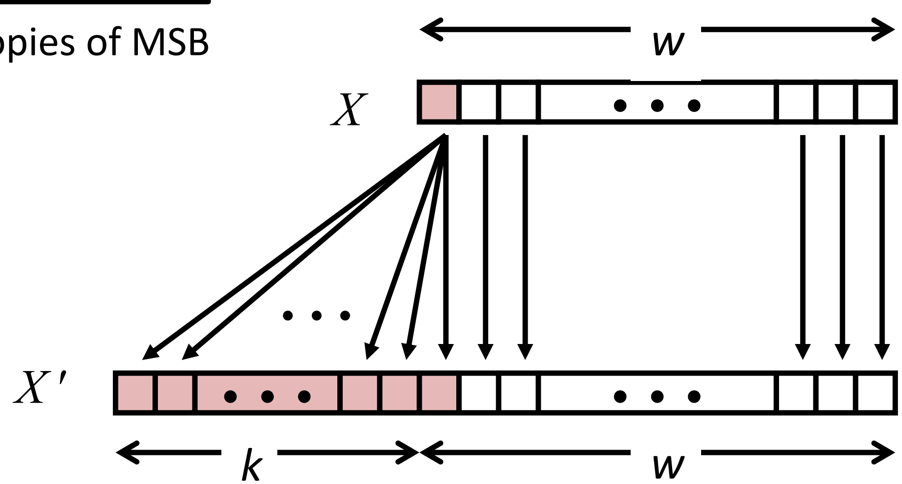
Summary
Casting Signed ↔ Unsigned: Basic Rules

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

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

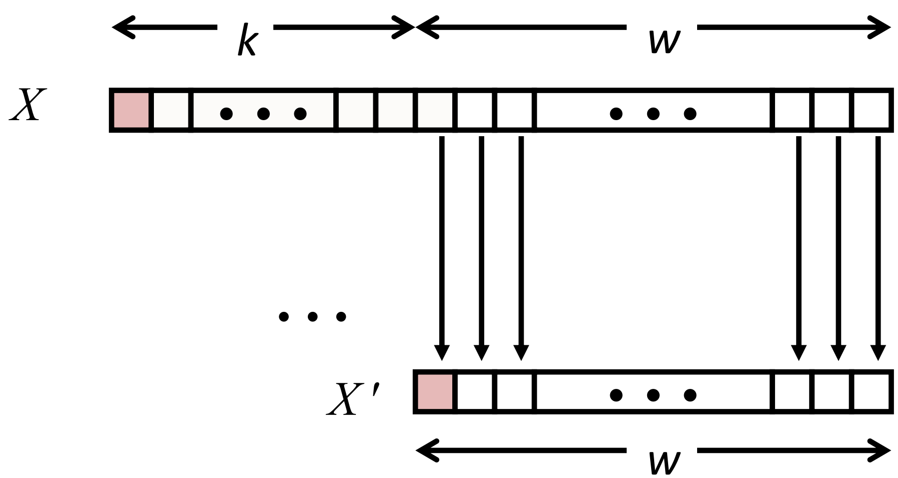
Sign Extension

- Task:
  - Given w-bit signed integer x
  - Convert it to $w+k$-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation

- Task:
  - Given k+w-bit signed or unsigned integer X
  - Convert it to w-bit integer X′ with same value for "small enough" X
- Rule:
  - Drop top k bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Summary: Expanding, Truncating: Basic Rules

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

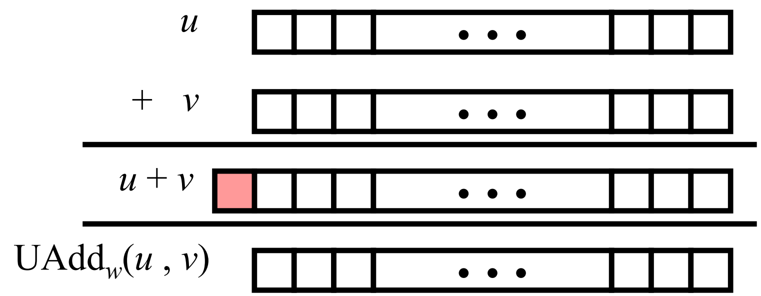
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

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

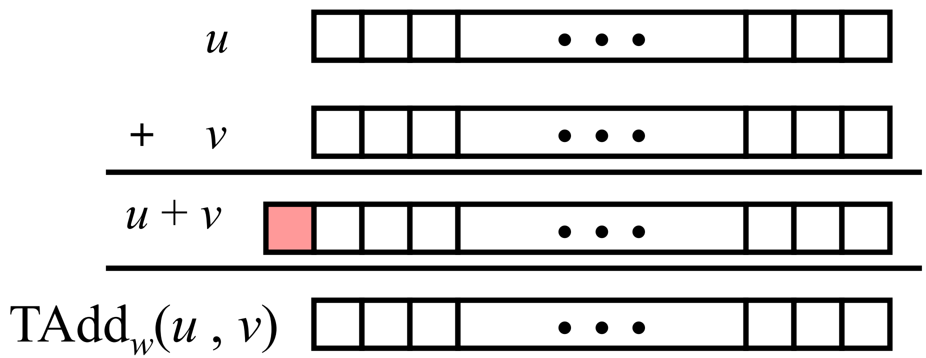
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Two's Complement Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



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

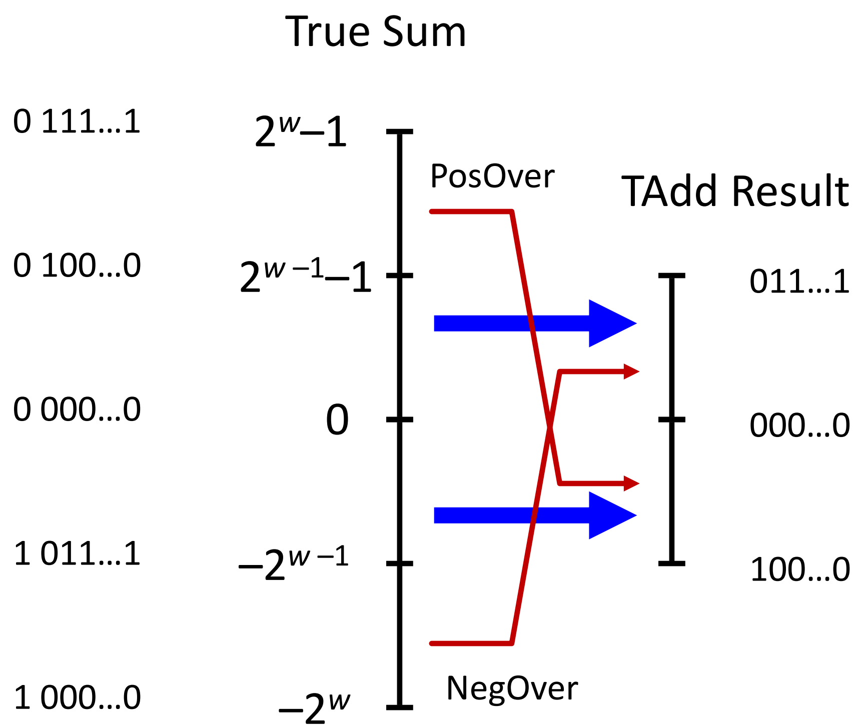
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

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

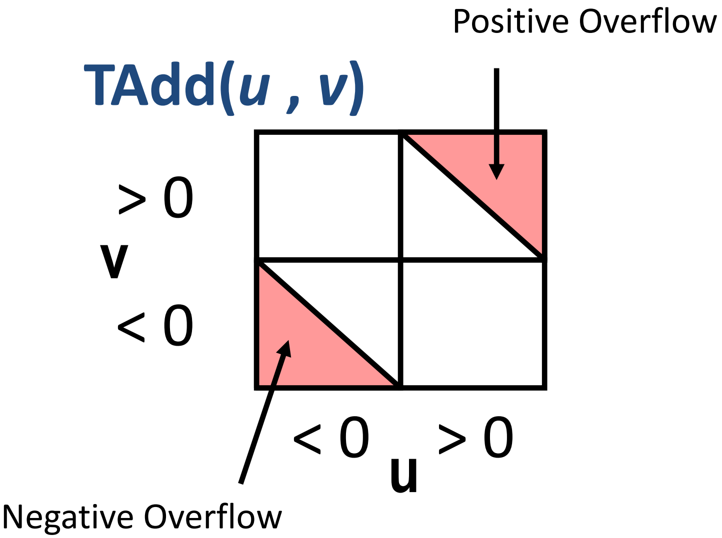
TAdd Overflow

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



Characterizing TAdd

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



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

Multiplication

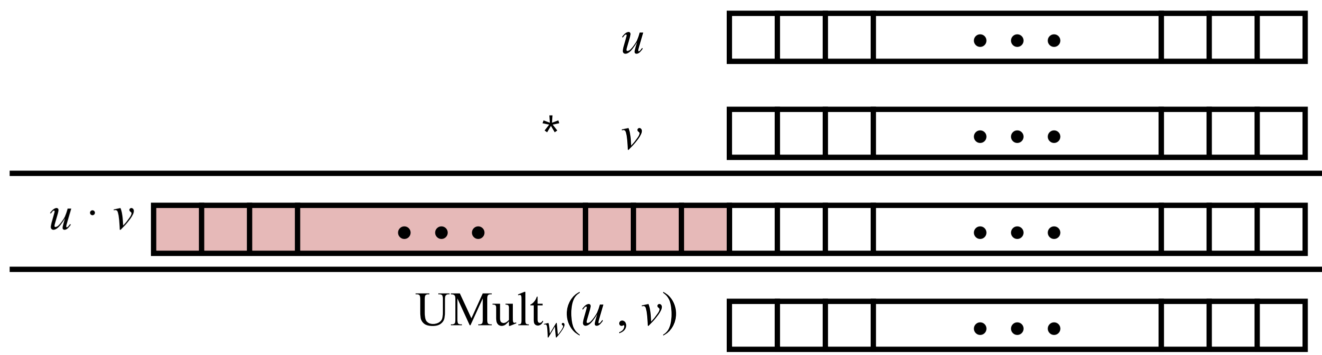
- Goal: Computing Product of w-bit numbers x, y
  - Either signed or unsigned
- But, exact results can be bigger than w bits
  - Unsigned: up to $2w$ bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to $2w$ 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

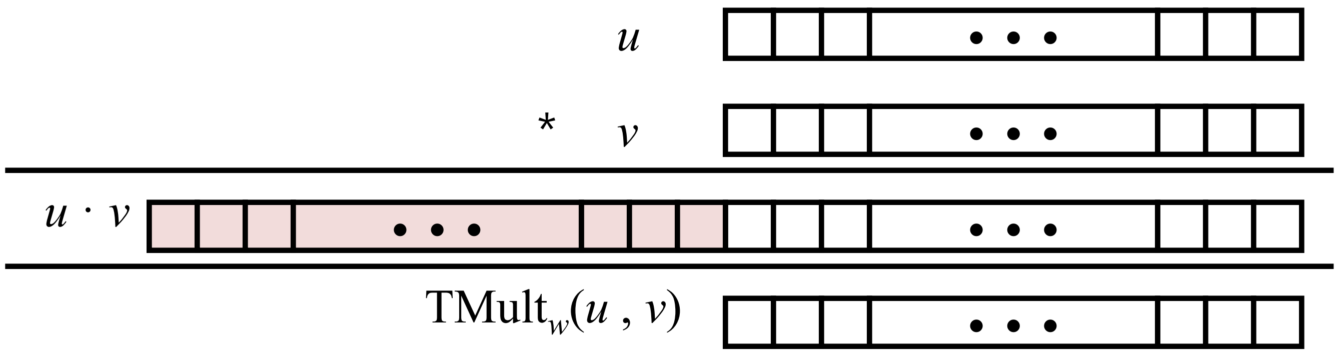
  $\mathrm{UMult}_w(u, v) = u \cdot v \bmod 2^w$

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

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

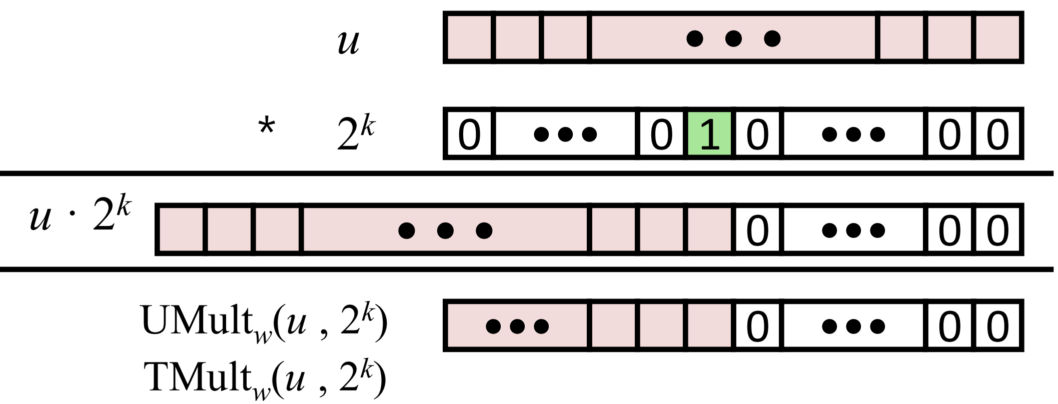
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w*+*k* bits

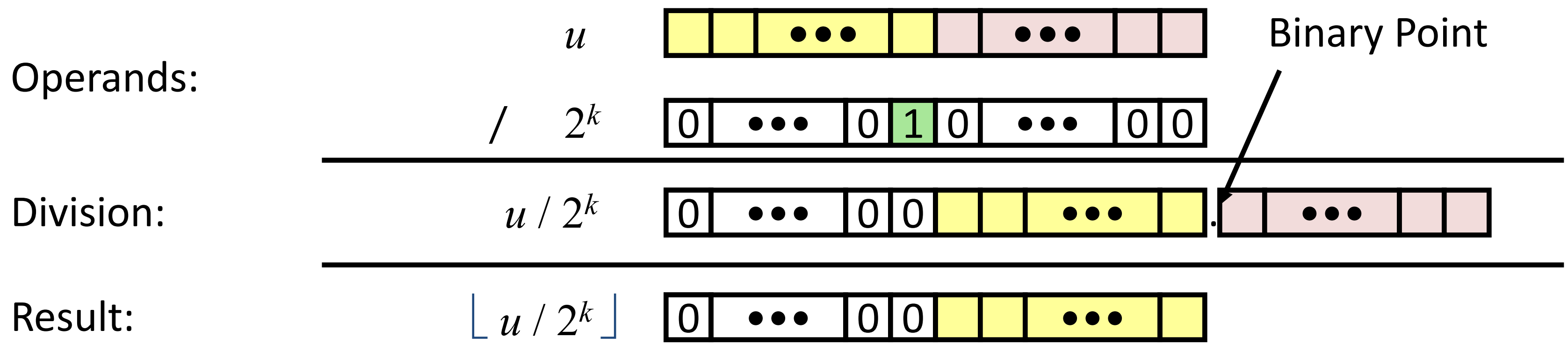
Discard *k* bits: *w* bits



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

Unsigned Power-of-2 Divide with Shift

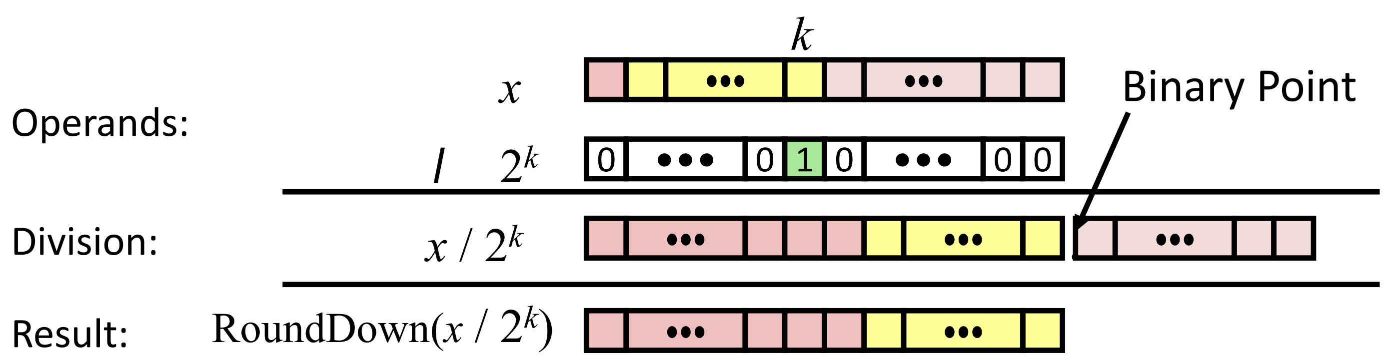
- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - Uses logical shift



	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`

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`

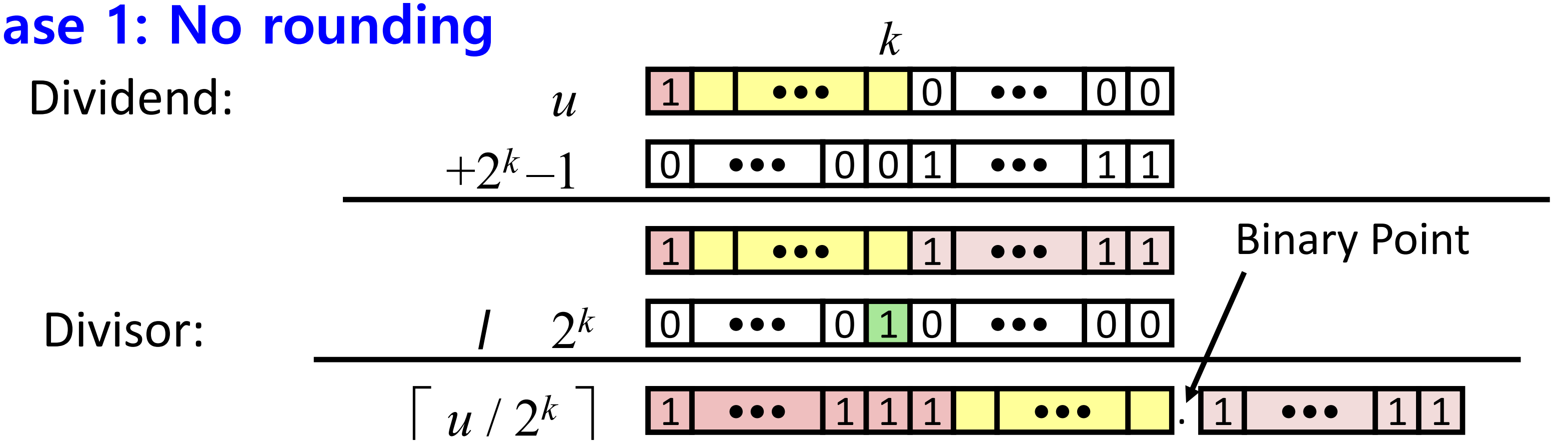


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want ⌈ **`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

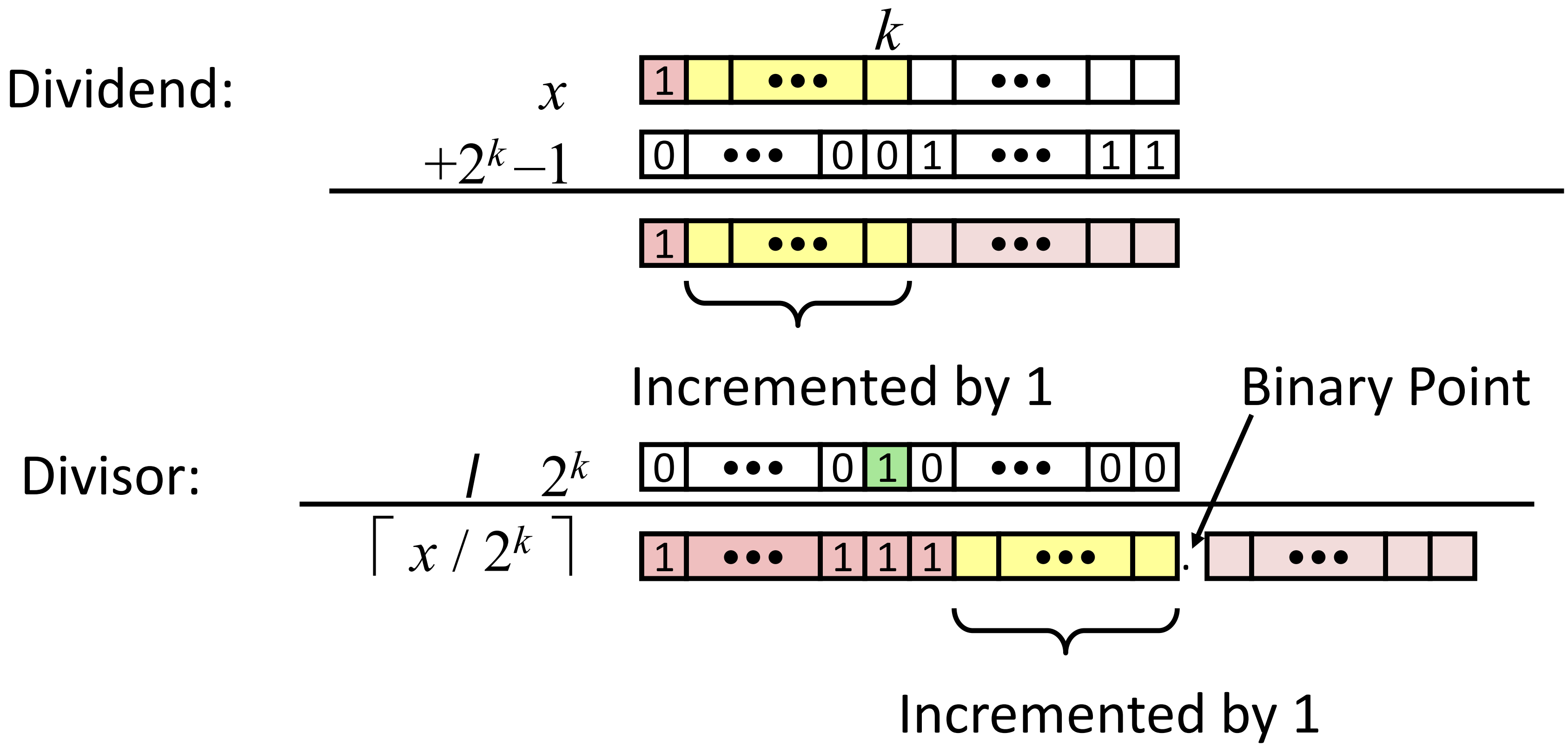
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Negation: Complement & Increment

- Negate through complement and increase
  `~x + 1 == -x`
- Example
  - 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

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`

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)

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

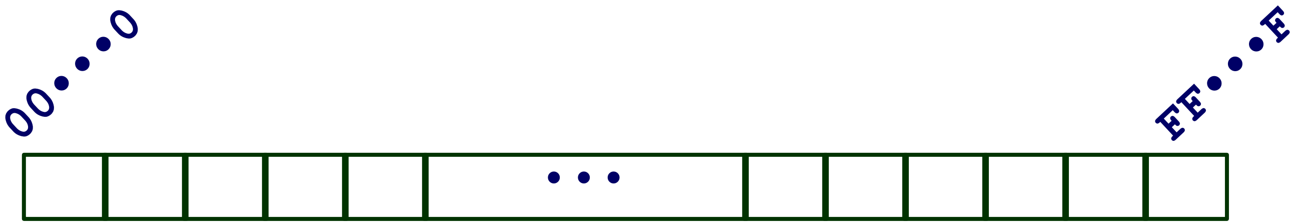
```
size_t i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

  - 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
- *Do* Use In System Programming
  - Bit masks, device commands,…

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

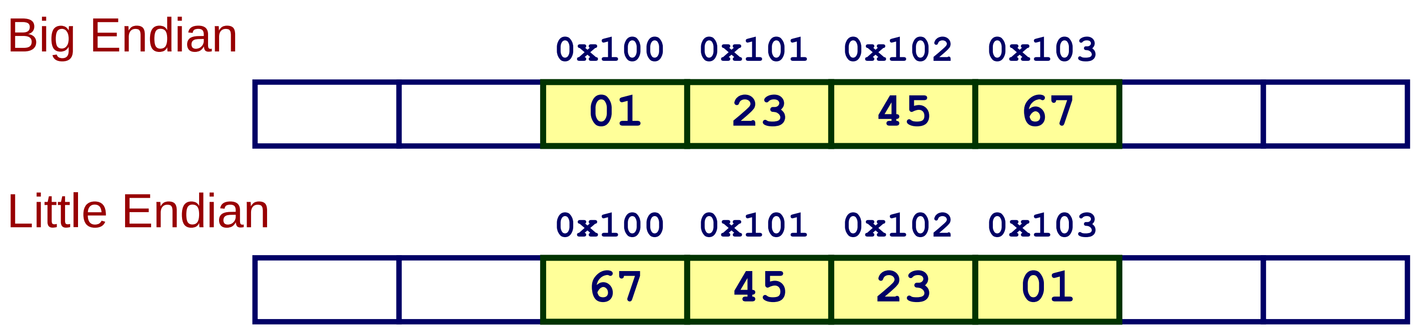
- Any given computer has a "Word Size"
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (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

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

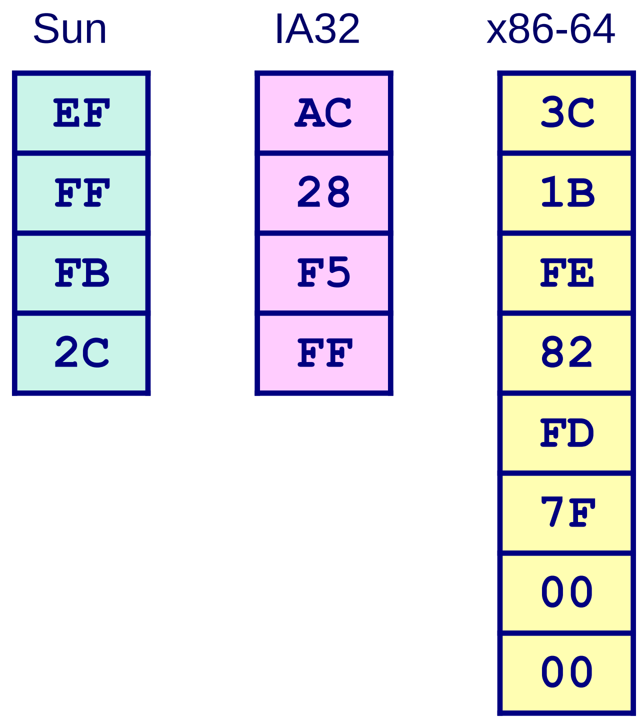
Byte Ordering Example

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



Representing Pointers

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



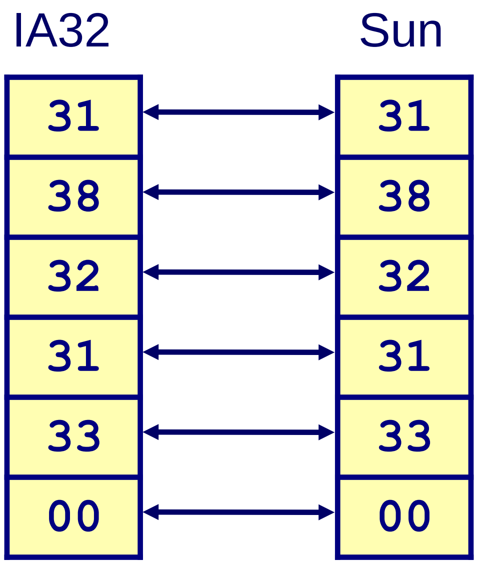
Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

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

- 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



Summary

- Integers
  - **Representation: unsigned and signed**
  - **Conversion, casting**
  - **Expanding, truncating**
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, string

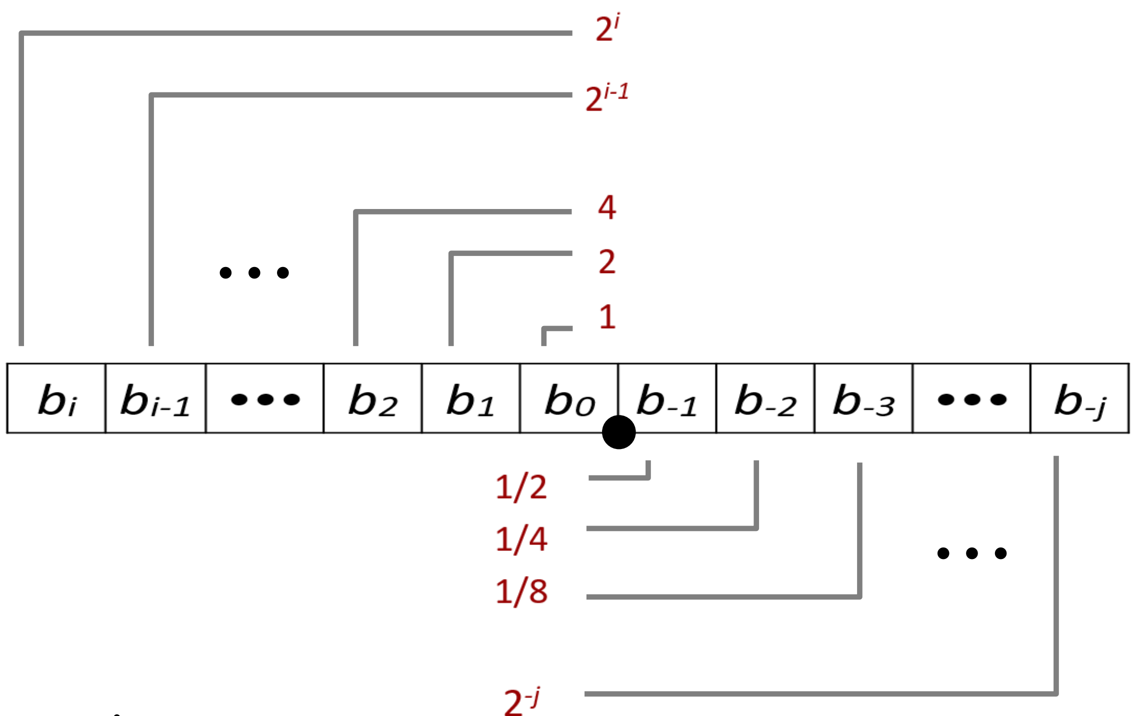
Outline

- Integers
- Floating points

Fractional binary numbers

- What is 1011.101_2?

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

Value	Representation	
5 3/4 = 23/4	101.11_2	= 4 + 1 + 1/2 + 1/4
2 7/8 = 23/8	10.111_2	= 2 + 1/2 + 1/4 + 1/8
1 7/16 = 23/16	1.0111_2	= 1 + 1/4 + 1/8 + 1/16

- 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
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - Value Representation
    - 1/3 **`0.0101010101[01]`**$\ldots_2$
    - 1/5 **`0.001100110011[0011]`**$\ldots_2$
    - 1/10 **`0.0001100110011[0011]`**$\ldots_2$

- 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
  - 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 fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

This is important!

- Ariane 5 explodes on maiden voyage: $500 MILLION dollars lost
  - 64-bit floating point number assigned to 16-bit integer
  - Causes rocket to get incorrect value of horizontal velocity and crash
  - https://www.youtube.com/watch?v=5tJPXYA0Nec
- Patriot Missile defense system misses scud – 28 people die
  - System tracks time in tenths of second
  - Converted from integer to floating point number.
  - Accumulated rounding error causes drift. 20% drift over 8 hours.
  - Eventually (on 2/25/1991 system was on for 100 hours) causes range mis-estimation sufficiently large to miss incoming missiles.

Floating Point Representation

- Numerical Form:

$$(-1)^s \; \boldsymbol{M} \;\; 2^{\boldsymbol{E}}$$

  - **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

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

- 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)

s	**exp**	**frac**

Precision options

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

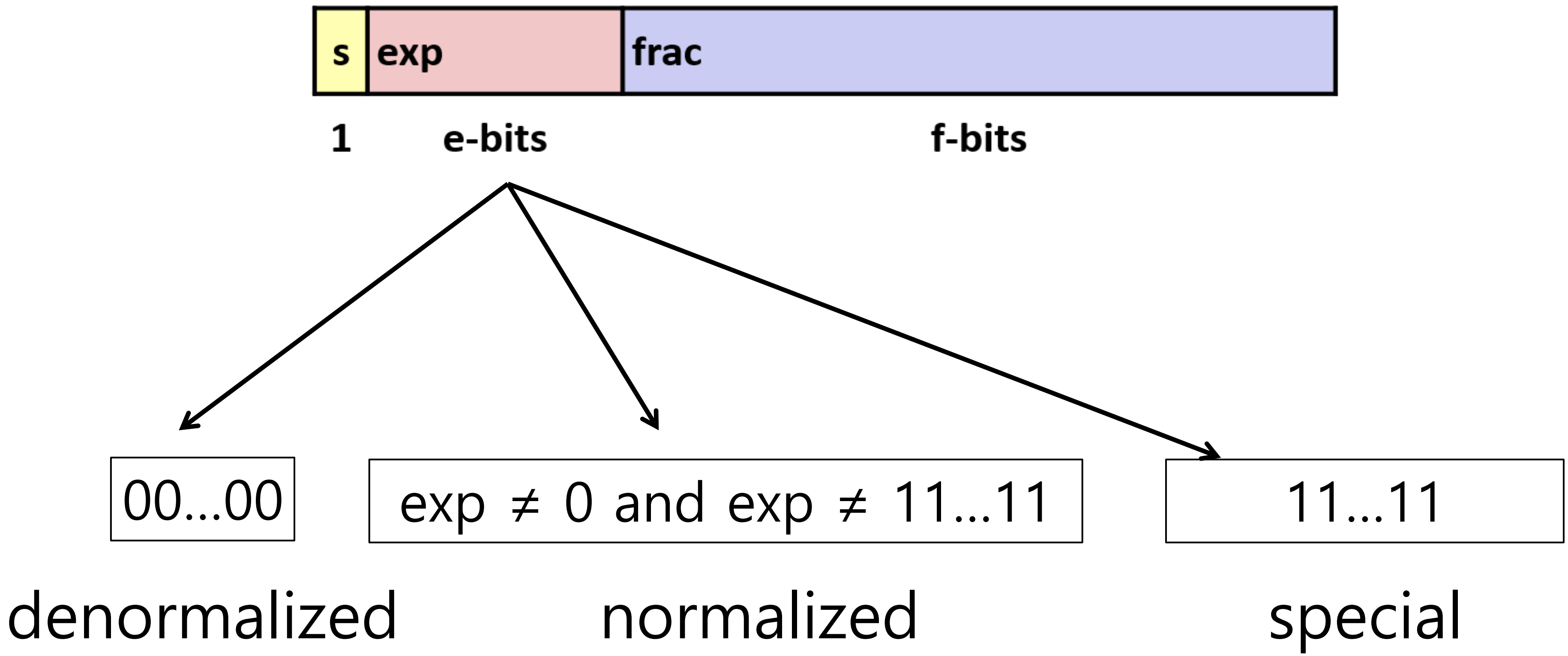
s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits
  ≈ 16 decimal digits, $10^{\pm 308}$

s	exp	frac
1	11-bits	52-bits

- Other formats: half precision, quad precision

Three “kinds” of floating point numbers



"Normalized" Values

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

- When: **`exp`** ≠ 000…0 and **`exp`** ≠ 111…1

- Exponent coded as a ***biased*** value: $\boldsymbol{E}$ = **`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: $\boldsymbol{M}$ = $1.xxx\ldots x_2$
  - 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 = $\texttt{1.}\underline{\texttt{1101101101101}}_2$
  frac= $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- Exponent
  E = 13
  $Bias$ = 127
  exp = 140 = $\texttt{10001100}_2$

- Result:

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

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

- Condition: exp = 000…0
- Exponent value: E = 1 – Bias (instead of **`exp`** – ***Bias***) (why?)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - **`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

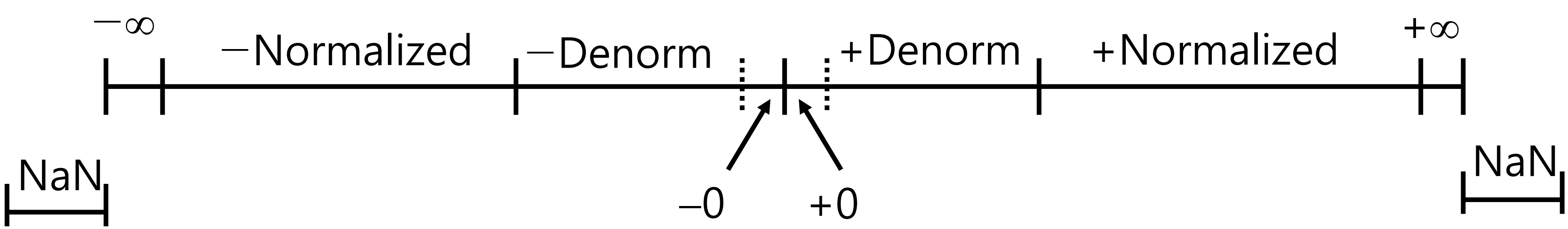
Special Values

- Condition: **`exp`** = **`111…1`**

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

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

Visualization: Floating Point Encodings



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? The answer is complicated.
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $\mathbf{x} +_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} + \mathbf{y})$
- $\mathbf{x} \times_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} \times \mathbf{y})$
- 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 ↓

*Round to nearest, but if half-way in-between then round to nearest even

Closer Look at Round-To-Even

- Default Rounding Mode
  - 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
- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

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

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is **`0`**
  - "Half way" when bits to right of rounding position = **`100…`**$_2$
- 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 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

`1.BBGRXXX`

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- Round up conditions
  - 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

- $(-1)^{s1}\ \boldsymbol{M1}\ \ 2^{\boldsymbol{E1}}\ \ \times\ \ (-1)^{\boldsymbol{s2}}\ \boldsymbol{M2}\ \ 2^{\boldsymbol{E2}}$
- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ \ 2^{\boldsymbol{E}}$
  - Sign *s*: $s1 \wedge s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$
- Fixing
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit **`frac`** precision
- Implementation
  - Biggest chore is multiplying significands

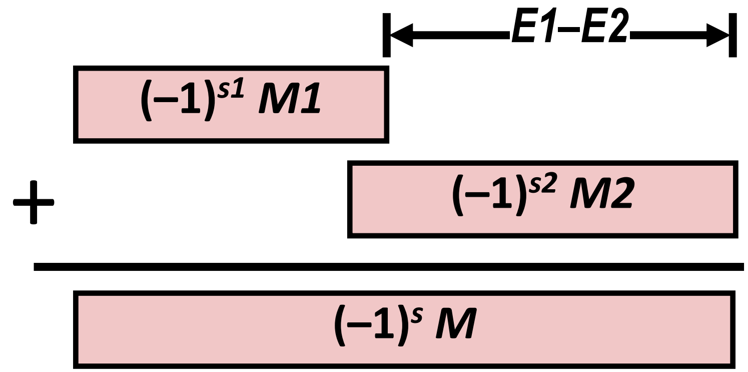
4 bit significand: $\texttt{1.010*2}^{2}\ \textbf{x}\ \texttt{1.110*2}^{3} = \texttt{10.0011*2}^{5}$

$= \texttt{1.00011*2}^{6} = \texttt{1.001*2}^{6}$

Floating Point Addition

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

Get binary points lined up



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

Mathematical Properties of FP Add

- 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
  - $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

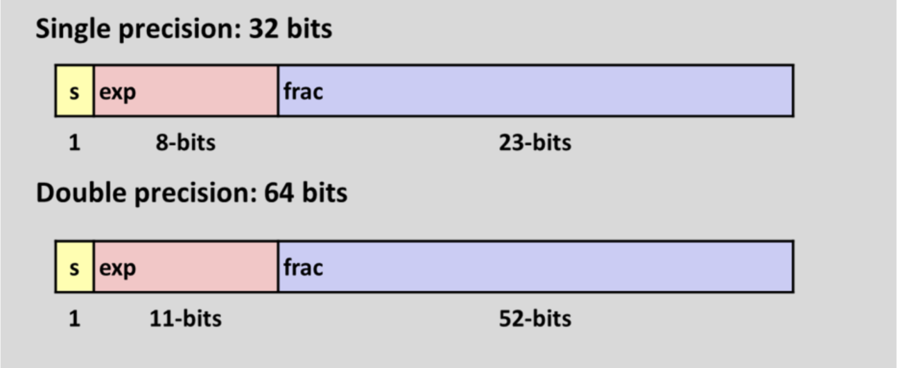
- Compare to Commutative Ring
  - Closed under multiplication? ***Yes***
    - But may generate infinity or NaN
  - Multiplication Commutative? ***Yes***
  - Multiplication is Associative? ***No***
    - Possibility of overflow, inexactness of rounding
    - Ex: **`(1e20*1e20)*1e-20= inf`**, **`1e20*(1e20*1e-20)= 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
  - **`double`** 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

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



Questions?