Lecture 3: Arithmetic

CS 105

January 30, 2019

Representing Integers

- unsigned:

$$\text{UnsignedValue}(x) = \sum_{j=0}^{w-1} x_j \cdot 2^j$$

- signed (two's complement):

$$\text{SignedValue}(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{j=0}^{w-2} x_j \cdot 2^j$$

Note: to compute –x for a signed int x, flip all the bits, then add 1

$$x + \sim x = 11\ldots1 = -1\text{, so } x + (\sim x + 1) = 0$$

Example: Three-bit integers

unsigned		signed
111	7	
110	6	
101	5	
100	4	
011	3	011
010	2	010
001	1	001
000	0	000
	-1	111
	-2	110
	-3	101
	-4	100

- The high-order bit is the *sign bit.*
- The largest unsigned value is $11\ldots1$, UMax.
- The signed value for -1 is always $11\ldots1$.
- Signed values range between `TMin` and `TMax`.

This representation of signed values is called *two's complement.*

Addition Example

- Compute 5 + 1 assuming all ints are stored as three-bit unsigned values

- Compute -3 + 1 assuming all ints are stored as three-bit signed values (two's complement)

Addition and Subtraction

- Usual addition and subtraction
  - Like you learned in second grade, only binary
  - Same for unsigned and signed
  - … but error conditions differ

Error Cases

- Unsigned addition:
  - $x +_w^u y = \begin{cases} x + y & \text{(normal)} \\ x + y - 2^w & \text{(overflow)} \end{cases}$
  - overflow has occurred iff $x +_w^u y < x$
- Signed addition:
  - $x +_w^t y = \begin{cases} x + y - 2^w & (positive\ overflow) \\ x + y & (normal) \\ x + y + 2^w & (negative\ overflow \end{cases}$
  - over flow has occurred iff $x > 0$ and $y > 0$ and $x +_w^t y < 0$
    or $x < 0$ and $y < 0$ and $x +_w^t y > 0$

Multiplication Example

- Compute 5 * 3 assuming all ints are stored as three-bit unsigned values

- Compute -3 * 3 assuming all ints are stored as three-bit signed values (two's complement)

Arithmetic, Part 2

- Multiplication
  - Product can be two words long; it may be truncated to one word
  - Bit level equivalence for unsigned and signed

Error Cases

- Unsigned multiplication:
  - $x *_w^u y = (x \cdot y) \bmod 2^w$
- Signed multiplication:
  - $x *_w^t y = U2T((x \cdot y) \bmod 2^w)$

Multiplying with Shifts

C uses << and >>. The arithmetic/logical choice is made according the the operands being signed/unsigned.

Java has no unsigned integers, but it has a third shift >>> for logical right shift.

We can multiply (often faster than with the processor's multiply instruction) with shifts.

- $$\begin{aligned} x \times 24 &= x \times 32 - x \times 8 \\ &= (x << 5) - (x << 3) \end{aligned}$$

Most compilers will generate this code automatically.

Signed Division by a Power of 2

- `x >> k` computes `x / 2`^{k}, rounded towards $-\infty$
- C on Intel processors rounds towards 0
  - `-11 >> 2 == -3`, but `-11/4 == -2`
- Solution: If x < 0, add 2^k-1 before shifting
  - Why does this work?

```
if (x < 0)
   x += (1 << k) - 1;
return x >> k;
```

Integer Types in C

- All integer types (char, short, int, long) can be prefixed with unsigned
- Constants are, by default, signed. Unsigned constants have the suffix U
- Casting between unsigned and signed changes the interpretation, but not the bits

Casting Types in C

- “Casting” means changing the type of a value

```
sometype x;
othertype y;

x = y;      // type error!

x = (sometype) y;
```

- Sometimes it means “interpret these bits in a different way”
    - Unsigned to/from signed
- Other times it means “convert these bits to the same value in a different representation”
    - Shorter integer types to/from longer
    - Integer types to/from floating point
- Implicit casting occurs in assignments and parameter lists. In mixed expressions, signed values are implicitly cast to unsigned
    - Source of many errors!

When to Use Unsigned

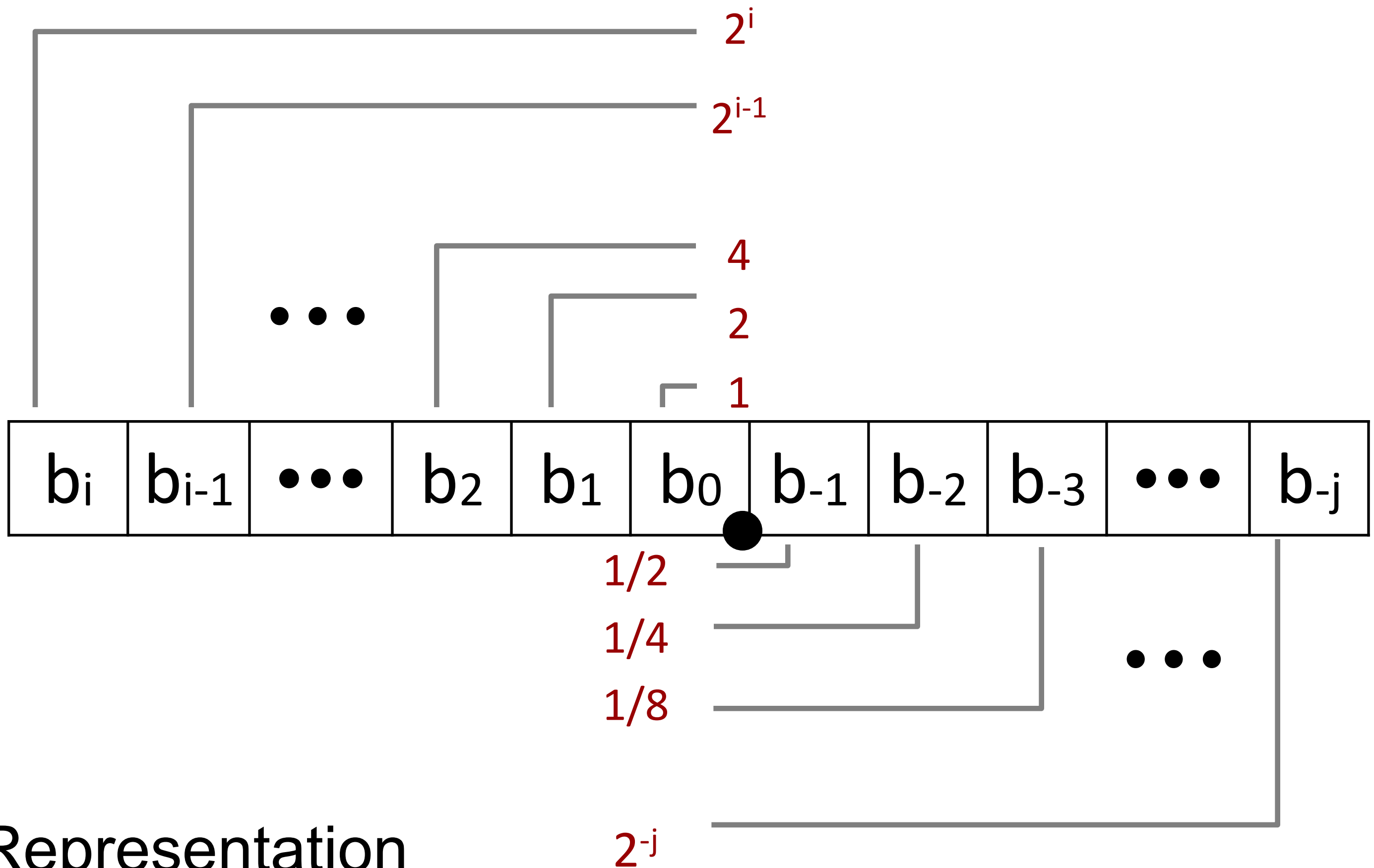
- Rarely
- When doing multi-precision arithmetic, or when you need an extra bit of range … but be careful!

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

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	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

- 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]`**…$_2$
    - 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?)

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
  - 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 (float): 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision (double): 64 bits

s	exp	frac
1	11-bits	52-bits

Normalized and Denormalized

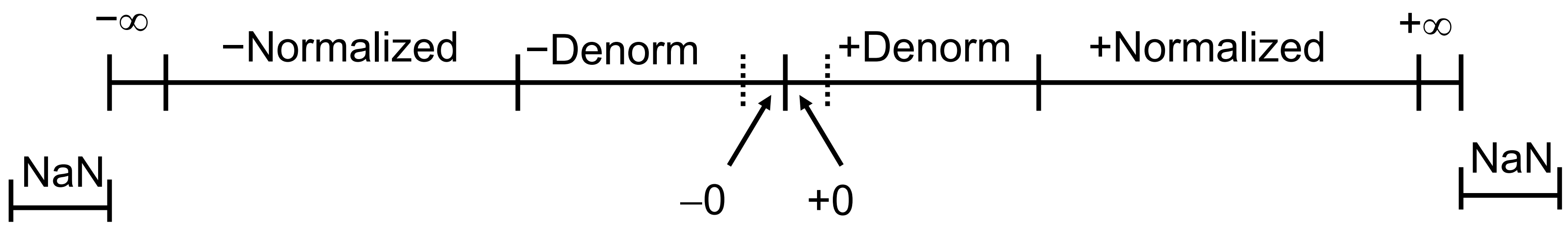
s	exp	frac
1	8-bits	23-bits

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

- Normalized Values
  - exp is neither all zeros nor all ones
  - normal case
  - exponent is defined as $\mathrm{E} = e_{k-1} \dots e_1 e_0 \; - \mathrm{bias}$, where $\mathrm{bias} = 2^k - 1$ (e.g., 127 for float or 1023 for double)
  - significand is defined as $M = 1.f_{n-1}f_{n-2} \dots f_0$

- Denormalized Values
  - exp is either all zeros or all ones
  - if all zeros: $\mathrm{E} = 1 - \mathrm{bias}$ and $M = 0.f_{n-1}f_{n-2} \dots f_0$
  - if all ones: infinity (if f is all zeros) or NaN

Visualization: Floating Point Encodings



Exercise

s	exp	frac
1	8-bits	23-bits

- Write a C function to compute a floating point representation of 2^x y directly constructing the IEEE float representation of the result. When x is too small, return 0.0 When x is too large, return $+\infty$

```
float fpwr2(int x){
  unsigned exp, frac, u;

  if(x<_______________){  /* Too small */
    exp = ____________;
    frac = ___________;
  } else if (x < _____){  /* Denormalized */
    exp = ____________;
    frac = ___________;
  } else if (x < _____){  /* Normalized */
    exp = ____________;
    frac = ___________;
  } else {  /* Too big */
    exp = ____________;
    frac = ___________;
  }

  u = ________________;  /* pack exp, frac */
  return u2f(u);  /* return as float */
}
```

Exercise

s	exp	frac
1	8-bits	23-bits

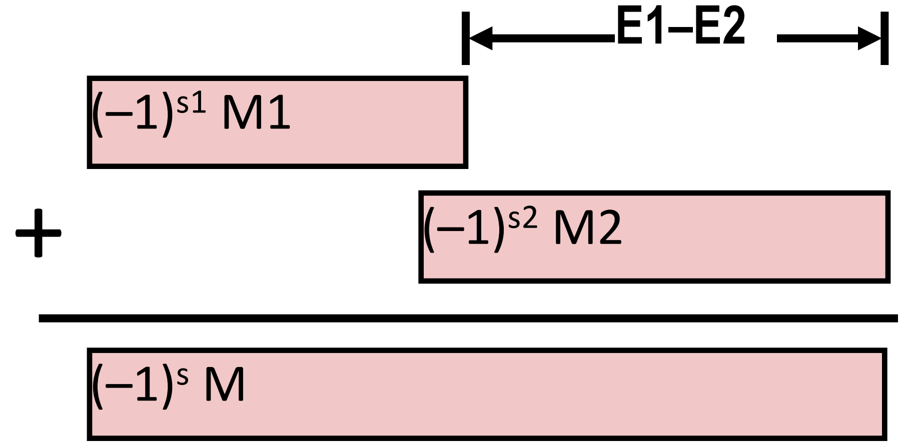
- Write a C function to compute a floating point representation of 2^x y directly constructing the IEEE float representation of the result. When x is too small, return 0.0 When x is too large, return +∞

```
float fpwr2(int x){
  unsigned exp, frac, u;

  if(x<-149){  /* Too small. -126-23=-149 */
    exp = 0;
    frac = 0;
  } else if (x < -126){  /* Denormalized */
    exp = 0;
    frac = 1 << (x+149);
  } else if (x < 128){  /* Normalized */
    exp = x+127;
    frac = 0;
  } else {  /* Too big. Return +infty */
    exp = 255;
    frac = 0;
  }

  u = exp << 23 | frac;  /* pack exp, frac */
  return u2f(u);  /* return as float */
}
```

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1} \quad + \quad (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume E1 > E2
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1
- Fixing
  - If M ≥ 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



FP Multiplication

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