Signed Numbers & Floating Point

CMSC 313

Raphael Elspas

Adding binary numbers

Ex 1) $23 + 18 = 00010111_2 + 00010010_2$

```
   1   11
 0001 0111
+0001 0010
----------
 0010 1001₂
```

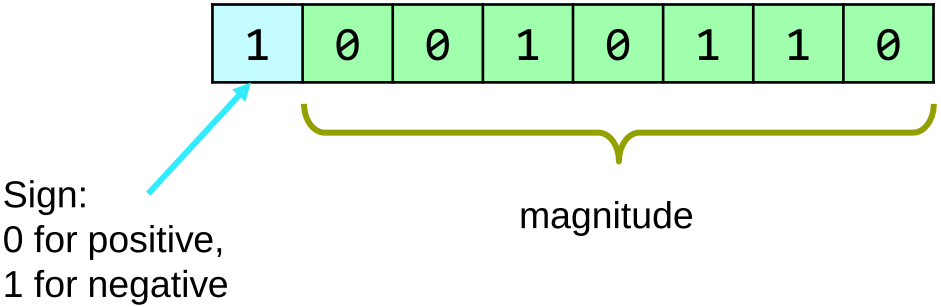
Ex 2) $89_{10} + 104_{10}$

```
 1111
 0101 1001
+0110 1000
----------
 1100 0001₂
```

How should we store negative numbers?

First try:

- Restrict the first bit to store sign
- In the rest, store the magnitude



Result: -22

Problems with sign + magnitude

- For an 8 bit number in binary, flipping the last bit would produce:

Decimal	Binary
-127	1111 1111
-126	1111 1110
…	…
-1	1000 0001
-0	1000 0000
+127	0111 1111
+126	0111 1110
…	…
+1	0000 0001
+0	0000 0000

- There are two 0s, which is inefficient
- Bad for adding numbers together

Second Try

- Invert each of the other bits to store a negative number
- The first bit will still store the sign.
- Ex: -11_{10} will be:

  `11`$_{10}$ `= 0000 1011`$_2$

  `~11`$_{10}$ `= 1111 0100`$_2$
- Called "**1's Complement**"

Decimal	Binary
-0	1111 1111
-1	1111 1110
…	…
-126	1000 0001
-127	1000 0000
+127	0111 1111
+126	0111 1110
…	…
+1	0000 0001
+0	0000 0000

Problems with 1's Complement

- Two zeros, inefficient storage
- While adding, you have to handle overflow: extra hardware
- Ex: 7 – 5

```
= 7 + (-5)

    7    0111
+  -5    1010
 ------------
       1 0001
+           1
 ------------
         0010
```

If there is a 1 overflow, add it to the output

Decimal	Binary
-0	1111 1111
-1	1111 1110
…	…
-126	1000 0001
-127	1000 0000
+127	0111 1111
+126	0111 1110
…	…
+1	0000 0001
+0	0000 0000

Third try's a charm

- Invert each bit & add 1.
- Adding 1 in beginning removes need for adding in overflow
- Called "**2's complement**"

Decimal	Binary
-1	1111 1111
-2	1111 1110
…	…
-127	1000 0001
-128	1000 0000
+127	0111 1111
+126	0111 1110
…	…
+1	0000 0001
+0	0000 0000

2’s complement bounds

- Since we are using 1 bit for sign, the range of values we can store changes.
- If x is an n-bit number in 2’s complement, it has the range:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

- If a number cannot be computed with a given number of bits, we say the number “**overflows**”

- NOTE: Overflow is different than carrying out from the final digit computation in 2’s complement

2's Complement overflow terminology

- This **IS overflow** : (8bit 2's-complement) $-100_{10} - 30_{10} = -130 < -2^7 = -128$
  (the answer is outside the bounds of the 2's complement 8bit range)

- This is a carry bit and **NOT overflow**:
  (the answer has a carry out of the MSB)

```
    23   0001 0111
+  -17   1110 1111
-------------------
       1 0000 0110
```

Detect 2's complement overflow

- If a **positive** plus a **positive** yields a **negative** result, overflow has occurred
- If a **negative** plus a **negative** yields a **positive** result, overflow has occurred
- Otherwise, the sum has not overflowed

A	B	A+B	Overflow?
+	+	-	Yes
-	-	+	Yes
Otherwise			No

8-bit 2's complement addition

Ex 1) $23_{10} - 17_{10} = 23 + (-17)$

```
17 = 0001 0001₂
     1110 1110 (inverted)
     1110 1111 (+1)

   23   0001 0111
+ -17   1110 1111
-----------------
      1 0000 0110
```

Truncate carry bit

Ans = $0000\ 0110_2$

Pos + Neg = Pos
NO OVERFLOW

Ex 2) $89_{10} - 104_{10} = 89 + (-104)$

```
104 = 0110 1000₂
      1001 0111 (inverted)
      1001 1000 (+1)

    89   0101 1001
+ -104   1001 1000
------------------
         1111 0001
```

Ans = $1111\ 0001_2$

Pos + Neg = Neg
NO OVERFLOW

2's complement overflow example

```
   -100 1001 0101
 + -30   1110 0010
 ----------------
      1 0111 0111
```

Truncate carry bit

```
Ans = OVERFLOW
```

Neg + Neg sum to a Pos -> OVERFLOW

```
   100 0110 0100
 +  30  0001 1110
 ----------------
        1000 0010
```

no carry bit

```
Ans = OVERFLOW
```

Neg + Neg sum to a Pos -> OVERFLOW

Summary of values

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Fractional numbers with scientific notation

- $110010110.10110_2 = 1.1001011010110 \times 2^8$

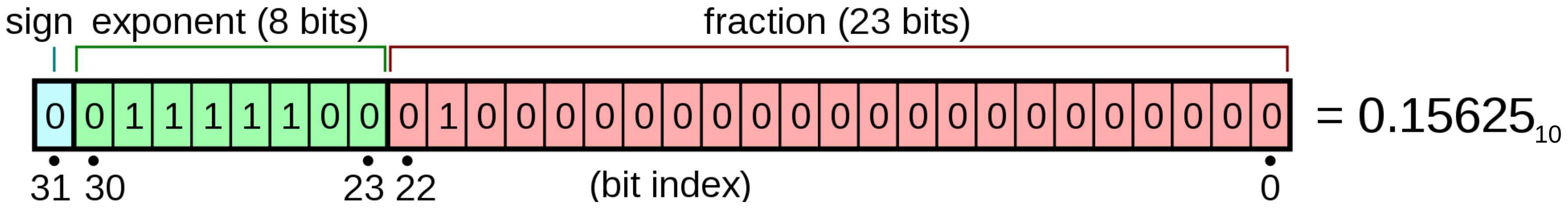
- **Exponent**: 8
- **Mantissa**: 0.1001011010110
- **Sign**: positive (0)

- Use these 3 parts to encode a floating point

IEEE 754 binary floating point (32 bit)

Three parts:

- Sign
- Exponent
- Mantissa (fractional part)

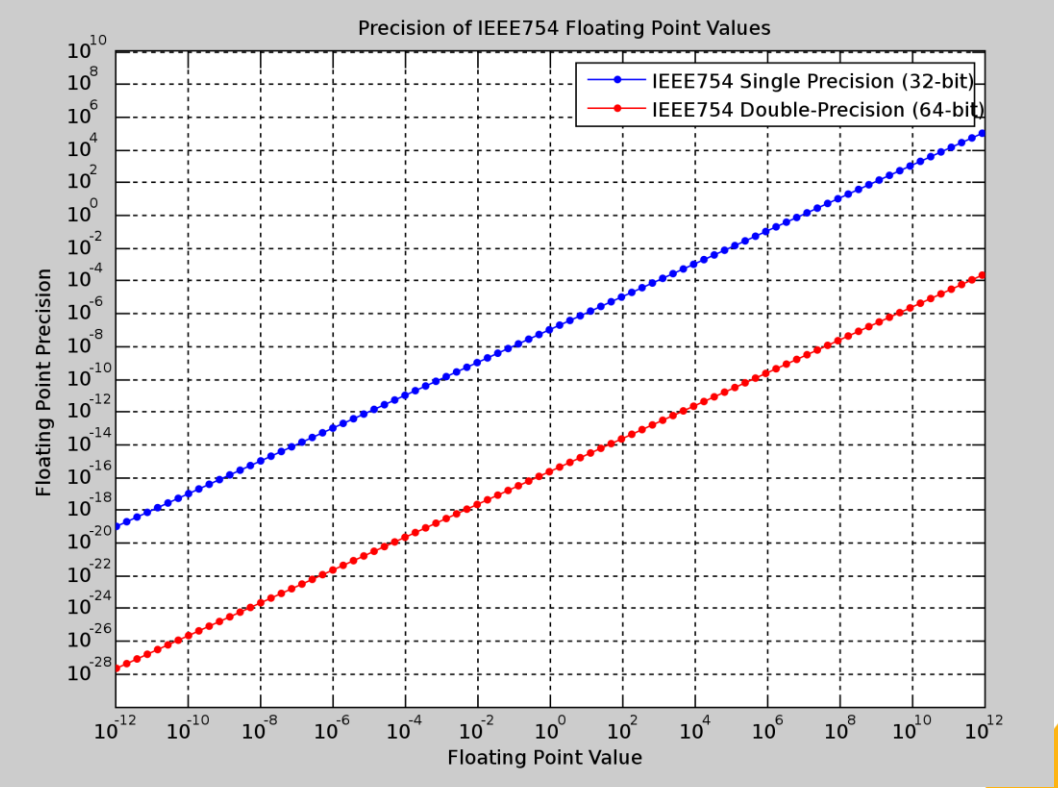


Floating point (32 bit) - exponent bias

- Want to store both positive and negative exponents so we can store large numbers, e.g. 23,057,183 and small numbers, e.g. 0.00000000032907
- We have 8 bits for the exponent, so our range for exponents will be -127 to +128.
- To calculate the bits in the exponent field, we add +127 to the exponent.
- E.g. if the number is $110010110.10110_2 = 1.1001011010110 \times 2^8$, then the exponent is 8, and the exponent field in IEEE754 float representation is $8 + 127 = 135_{10}$, or 10000111_2.

Precision (Error)

- As the number gets larger, more digits in the mantissa store whole number values instead of fractional values
- The mantissa must be truncated to clip off the least significant bits and store the most significant bits instead



Store a binary number in IEEE 754 Float (32 bit)

Ex: 110010110.10110_2

1. Convert to scientific notation:

   $1.100101101011 0 \times 2^8$

2. Find sign, mantissa and exponent

   sign: 0, mantissa: 0.1001011010110, exponent: 8

3. Adjust exponent with bias

   $8+127 = 135_{10} = 10000111_2$

4. Append all digits

sign	exp								mantissa																						
0	1	0	0	0	0	1	1	1	1	0	0	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
31	30							24	23																						0

Store a base 10 number in IEEE 754 Float (32 bit)

Ex: 263.3_{10}

1. Convert base-10 to binary:

   $263_{10} = 100000111$

   $0.3_{10} = 0.01\overline{0011}$
2. Find mantissa, sign and exponent: $1.0000011101\overline{0011}$
3. Adjust exponent with bias: $8+127 = 135_{10} = 10000111_2$
4. Append all digits

sign	exp								mantissa																						
0	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0
31	30							24	23																						0

Summary

- Adding binary numbers, 1+1 = 0, carry the 1.
- 1s complement: invert all bits
- 2s complement: invert all bits + add 1
    - Overflow happens when pos + pos = neg or neg + neg = pos
- Floating point: stored with sign, mantissa, and exponent
    - Sign: positive = 0, negative = 1
    - Exponent is biased by +127

References

- https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/
- https://en.wikipedia.org/wiki/IEEE_754
- https://www.youtube.com/watch?v=8afbTaA-gOQ
- https://www.h-schmidt.net/FloatConverter/IEEE754.html