Today

- Binary addition
- Representing negative numbers

Binary Addition

Consider the following binary numbers:

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

How do we add these numbers?

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

1

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

↓

0 1

And we have a carry now!

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

0 0 1

And we have a carry again!

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

↓

0 0 0 1

and again!

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

↓

1 0 0 0 1

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1

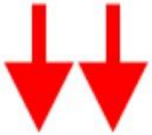
0 1 0 0 0 1

One more carry!

Binary Addition

0 0 1 0 0 1 1 0

0 0 1 0 1 0 1 1



0 1 0 1 0 0 0 1

Binary Addition

Behaves just like addition in decimal, but:

- We carry to the next digit any time the sum of the digits is 2 (decimal) or greater

Negative Numbers

So far we have only talked about representing non-negative integers

- What can we add to our binary representation that will allow this?

Representing Negative Numbers

One possibility:

- Add an extra bit that indicates the sign of the number
- We call this the “sign-magnitude” representation

Sign Magnitude Representation

+12 0 0 0 0 1 1 0 0

Sign Magnitude Representation

+12	0 0 0 0 1 1 0 0
-12	1 0 0 0 1 1 0 0

Sign Magnitude Representation

+12	0 0 0 0 1 1 0 0
-12	1 0 0 0 1 1 0 0

What is the problem with this approach?

Sign Magnitude Representation

What is the problem with this approach?

- Some of the arithmetic operators that we have already developed do not do the right thing

Sign Magnitude Representation

Operator problems:

- For example, we have already designed a counter (that implements an ‘increment’ operation)

-12 1 0 0 0 1 1 0 0

Sign Magnitude Representation

Operator problems:

-12 1 0 0 0 1 1 0 0

↓ Increment

Sign Magnitude Representation

Operator problems:

-12 1 0 0 0 1 1 0 0

↓ Increment

1 0 0 0 1 1 0 1

Sign Magnitude Representation

Operator problems:

-12 1 0 0 0 1 1 0 0

Increment

-13 ← 1 0 0 0 1 1 0 1

!!!!

Representing Negative Numbers

An alternative:

(a little intuition first)

0 0 0000000

Decrement

Representing Negative Numbers

An alternative:

(a little intuition first)

0	0 0000000	
	↓	Decrement
	1 1111111	

Representing Negative Numbers

An alternative:

(a little intuition first)

0   0 0 0 0 0 0 0 0

Decrement

Define this as

-1 ← 1 1 1 1 1 1 1 1

Representing Negative Numbers

A few more numbers:

3	0 0 0 0 0 0 1 1
2	0 0 0 0 0 0 1 0
1	0 0 0 0 0 0 0 1
0	0 0 0 0 0 0 0 0
-1	1 1 1 1 1 1 1 1
-2	1 1 1 1 1 1 1 0
-3	1 1 1 1 1 1 0 1

Two’s Complement Representation

In general, how do we take the additive inverse of a binary number?

Two's Complement Representation

In general, how do we take the additive inverse of a binary number?

- Invert each bit and then add '1'

Two's Complement Representation

Invert each bit and then add '1'

5	0 0 0 0 0 1 0 1	
↓	↓	Two's complement
-5	1 1 1 1 1 0 1 1	

Two's Complement Representation

Now: let's try adding a positive and a negative number:

12	0 0 0 0 1 1 0 0
+	+
-5	1 1 1 1 1 0 1 1

Two's Complement Representation

Now: let's try adding a positive and a negative number:

12	0 0 0 0 1 1 0 0
+	+
-5	1 1 1 1 1 0 1 1
	↓
	0 0 0 0 0 1 1 1

Two's Complement Representation

Now: let's try adding a positive and a negative number:

12	0 0 0 0 1 1 0 0
+	+
-5	1 1 1 1 1 0 1 1
7 ←	0 0 0 0 0 1 1 1

Two's Complement Representation

Two's complement is used for integer representation in today's processors

Two's Complement Representation

Two's complement is used for integer representation in today's processors

One oddity: we can represent one more negative number than we can positive numbers

Implementing Subtraction

How do we implement a ‘subtraction’ operator?

(e.g., A – B)

Implementing Subtraction

How do we implement a ‘subtraction’ operator?

(e.g., A – B)

- Take the 2s complement of B
- Then add this number to A

Representing Fractions

Floating point representations are expensive:

- Require many bits
- Either require specialized hardware or long functions to compute mathematical operations

A Low-Cost Alternative: Fixed Point Representations

"w.f" fixed point:

- w bits to represent the whole number (including the sign)
- f bits to represent the fraction

A Low-Cost Alternative: Fixed Point Representations

"w.f" fixed point:

- We are representing values in units of 2^{-f}

So: 5.3 fixed point

- 5 bits for whole

A Low-Cost Alternative: Fixed Point Representations

5.3 fixed point (fits in an int8_t)

- 5 bits for whole
- 3 bits for fraction

What can we represent with this?

A Low-Cost Alternative: Fixed Point Representations

What can we represent with 5.3 fixed point?

- 5 bits for whole: 15 … -16
- 3 bits for fraction: units of 1/8th

Fixed-Point Example

Fixed Point	Value	# of eighths
00000 000	0.0	0 eighths
00000 001		
00000 100		
00001 000		
00101 010		

Fixed-Point Example

Fixed Point	Value	# of eighths
00000 000	0.0	0 eighths
00000 001	0.125	1 eighth
00000 100	0.5	4 eighths
00001 000	1.0	8 eighths
00101 010	5.25	42 eighths

Adding Fixed-Point Numbers

```
int8_t a = 5;     // 5/8
int8_t b = 10;    // 10/8
int8_t c = a + b ???
```

$5\left(\frac{1}{8}s\right) + 10\left(\frac{1}{8}s\right) = 15$ what?

Adding Fixed-Point Numbers

```
int8_t a = 5;     // 5/8
int8_t b = 10;    // 10/8
int8_t c = a + b; // 15/8
```

$$5\left(\frac{1}{8}s\right) + 10\left(\frac{1}{8}s\right) = 15\left(\frac{1}{8}s\right)$$

So: addition does the right thing

Multiplying Fixed-Point Numbers

```
int8_t a = 5;     // 5/8
int8_t b = 10;    // 10/8
int8_t c = a * b ???
```

$5\left(\frac{1}{8}s\right) \times 10\left(\frac{1}{8}s\right) = 50$ what?

Multiplying Fixed-Point Numbers

```
int8_t a = 5;     // 5/8
int8_t b = 10;    // 10/8
int8_t c = a * b ???
```

$$5\left(\frac{1}{8}s\right) \times 10\left(\frac{1}{8}s\right) = 50\left(\frac{1}{64}s\right)$$

But: we need to keep things in 5.3 format

Multiplying Fixed-Point Numbers

```
int8_t a = 5;      // 5/8
int8_t b = 10;     // 10/8
int8_t c = (a * b) >> 3;  // 6/8
```

$$5\left(\frac{1}{8}s\right) \times 10\left(\frac{1}{8}s\right) = 50\left(\frac{1}{64}s\right) \approx 6\left(\frac{1}{8}s\right)$$

Dividing Fixed-Point Numbers

```
int8_t a = 20;        // 20/8
int8_t b = 7;         // 7/8
int8_t c = a / b ???
```

$$20\left(\frac{1}{8}s\right) \div 7\left(\frac{1}{8}s\right) = 2$$ What?

Dividing Fixed-Point Numbers

```
int8_t a = 20;     // 20/8
int8_t b = 7;      // 7/8
int8_t c = a / b ???
```

$$20\left(\frac{1}{8}s\right) \div 7\left(\frac{1}{8}s\right) = 2 \ \ (1s)$$

But: we want to stay within the 5.3 format. And – note that we have lost information in the rounding!

Dividing Fixed-Point Numbers

```
int8_t a = 20;      // 20/8
int8_t b = 7;       // 7/8
int8_t c = (a << 3) / b; // 160/7
```

$$20\left(\frac{1}{8}s\right) \div 7\left(\frac{1}{8}s\right) = 22\left(\frac{1}{8}s\right)$$

Notes About the Book

The example code that the book gives tries to address some additional questions (but fails to be clear):

- In conversions from floating point to fixed-point, it catches errors when a floating point value is too small or too large to fit in the fixed point representation
- assert(0) just means that an error should be generated

Notes About the Book

- In the book, a "short" is 16 bits and a "long" is 32 bits.
- For many of the fixed-point examples, the fixed-point values fit in 16 bits
- After we perform a mathematical operation, it is possible that the result will not fit within the 16 bits
- So: all numbers are converted to 32 bits before the operation & the results are checked before converting back to 16 bits