Floating-Point Numbers

Schedule

- **Today**
  - Finish up Floating-Point Numbers
  - **Homework #3 assigned**
- **Monday**
  - Character representation
  - **Homework #2 due**
  - **Quiz #1**
- **Wednesday**
  - Boolean Algebra / Logic
  - **Homework #3 due**

Quiz #1

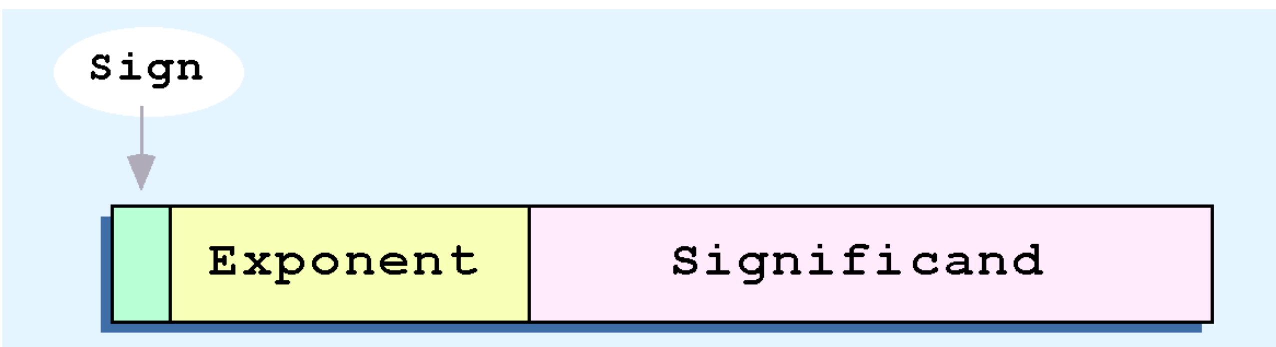
- **Topics from Homework 1 and 2**
- Conversion between decimal and binary
  - Whole numbers and fractional numbers
- Signed numbers
  - Sign-magnitude
  - 1's complement
  - 2's complement
- Conversion between hexadecimal and binary

Quiz #1

- **Topics from introductory lectures**
- History of computers
  - Vacuum tubes versus transistors?
  - Transistors versus integrated circuits?
- Moore's Law
  - What does it mean?
  - How far will it go?
- Basic computer operation
  - Von Neumann model
  - Basic instruction cycle
    - Fetch (from where?)
    - Decode (what?)
    - Execute
  - Key components
    - What is stored in memory?
    - What does the ALU do?

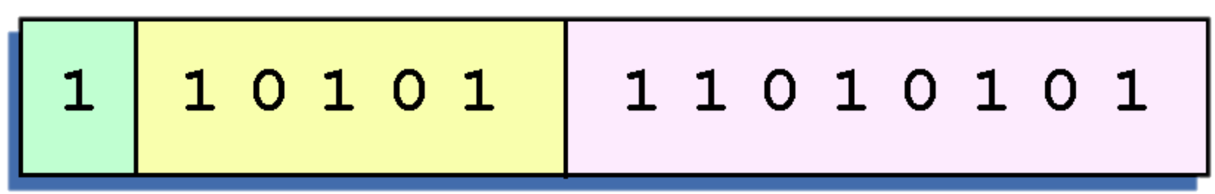
Recap – Floating-Point Representation

- "Simple Model"
- 14 bit long floating-point number:
  - The sign field is 1 bit
  - The exponent field is 5 bits
  - The significand field is 8 bits



Recap – Floating-Point Representation

- Example: Express **-26.625_{10}** in the revised 14-bit floating-point model
- $26.625_{10} = 11010.101_2 \times 2^0$
  Normalize = 0.11010101×2^5.
- Use excess 16 biased exponent:
  - $16 + 5 = 21_{10}$ ($=10101_2$)
- Also need a 1 in the sign bit (negative number)
- Final value saved to memory:



Biased Exponent – Why?

- **Why does the simplified model** (and the real standard, described next) **use a biased exponent** to store positive/negative numbers, instead of two's complement numbers?
- Only answer I could find:
  - It makes comparing two floating-point numbers faster, **even on integer hardware** that doesn't understand floating-point fields
  - Most significant bit – sign bit
  - Next most significant group – Exponents, which are perfectly arranged in ascending order, even for "negative" exponents

Floating-Point Representation

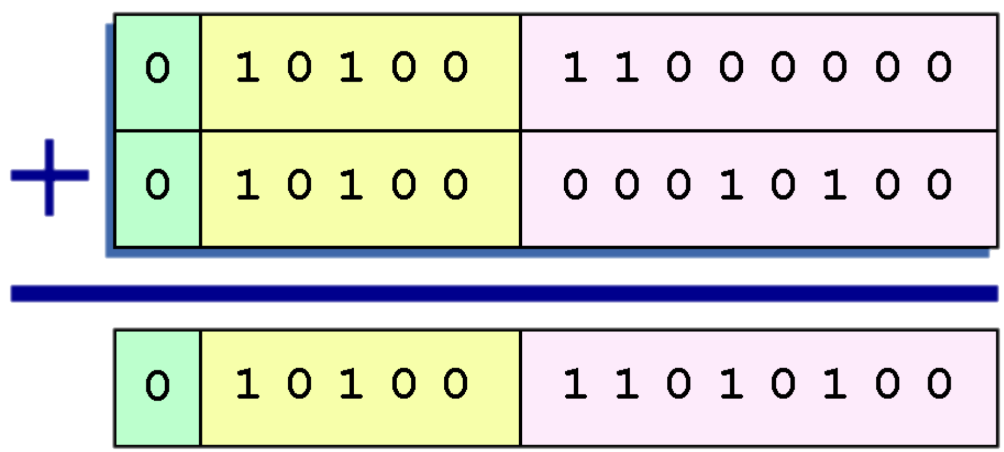
- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum
- If the exponent requires adjustment, we do so at the end of the calculation

Floating-Point Representation

- Example: Find 12_{10} + 1.25_{10} using the 14-bit simple floating-point model

- $12_{10} = 0.1100 \times 2^4$
  $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$

- Thus, the sum is 0.110101×2^4



Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
  - Multiply the two significands
  - Add their exponents
- If the exponent requires adjustment, do so at the end of the calculation

The **Real** Floating-Point Model

IEEE Floating-Point Representation

- The IEEE has established standards for floating-point numbers
- **IEEE-754 single precision** standard (**32 bits** long)
  - **8**-bit exponent (with a **bias of 127**)
  - **23**-bit significand
  - A “float” in C++
- **IEEE-754 double precision** standard (**64 bits** long)
  - **11**-bit exponent (with a **bias of 1023**)
  - **52**-bit significand
  - A “double” in C++

IEEE Floating-Point Representation

- **Watch out! Significand is normalized differently**
  - **Implied 1** to the **left** of the radix point, i.e. formatted as 1.xxxxxxx…
  - For example, $4.5 = .1001 \times 2^3$
    In IEEE format, use $4.5 = 1.001 \times 2^2$
  - The 1 is **implied**, which means it is **not saved in computer memory**
    - The stored significand would include only 001
    - Optimization – This saves one entire bit!

IEEE Floating-Point Representation

- Example: Express **-3.75** as a floating point number using IEEE *single* precision.
- Normalize according to IEEE rules:
  - $-3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias for *single precision* is 127, so add 127 + 1 = 128
    - This is the exponent saved to computer memory
  - The first 1 in the significand is implied, so we have:

1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(implied 1. not saved)

  - To decode saved number with the implied 1 in the significand:
    - $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

IEEE Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.
- Using the double precision standard:
  - An exponent of 2047 indicates a special value

IEEE Floating-Point Representation

- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1
- Programmers should avoid testing a floating-point value for equality to zero
  - Negative zero does not equal positive zero

Floating-Point Errors

Floating-Point Errors

- No matter how many bits we use in a floating-point representation, our model is finite
- Problem: Real numbers can be infinite, so our model can only *approximate* a real value
- At some point, every model breaks down, introducing errors into the calculations
- By using a greater number of bits in the model, we can reduce these errors, but we can never totally eliminate them

Floating-Point Errors

- Example: The 14-bit model cannot exactly represent the decimal value 128.5
  - In binary, it is 9 bits wide: $10000000.1_2 = 128.5_{10}$
  - But we only have an 8-bit significand!

Floating-Point Errors

- How much error occurs when 128.5_{10} is represented with the 14-bit model?
  - True number: 128.5
  - Approximated number: 128
  - Error (percent difference)

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

Floating-Point Errors

- If you wrote a loop that repetitively added 0.5 to 128.5 using 14-bit floating point, you would have an error of **nearly 2%** after only four iterations
  - *The error is less with "real" 32/64-bit floating point standards, but still exists*

Errors accumulate on real systems:

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
  printf("Floating-Point Demo Program\n");

  double a = 0.0;
  int i;

  for(i=0; i<1000000000; i++)
    {
      a = a + 0.1;
    }

  printf("A=%lf\n", a);

  return;
}
```

Actual output on test Linux machine:

```
Floating-Point Demo Program
A=99999998.745418
```

This revised program doesn't accumulate errors as quickly:

```
a=1000000000*0.1;
printf("A=%lf\n", a);
```

Actual output on test Linux machine:

A=100000000.000000

Another Demo Program:

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
  printf("Floating-Point Demo Program 1\n");
  double a, b;

  a = (58.0/40.0-1.0);
  b = (18.0/40.0);

  printf("A=%lf\n", a);
  printf("B=%lf\n", b);

  if(a==b)
    printf("A equals B!\n");
  else
    printf("A does not equal B!\n");

  return;
}
```

- **What will be the output of this program?**

Floating-Point Errors

- **Actual program output:**

```
jshafer@ecs-network:~/testing_ecpe170$ ./test
Floating-Point Demo Program 1
A=0.450000
B=0.450000
A does not equal B!
```

Wait, why??

Floating-Point Errors

- Digging deeper via C hackery…
- First get a pointer to memory location A
- Then copy the data at that location to a new 64-bit variable
- Then print it out in 16 hex digits
  - 16 hex = 64 bits

```
// Why aren't they equal?
// Let's dig into the contents of memory
uint64_t* ptr;
uint64_t myvalue;

ptr = (uint64_t*)&a;
myvalue=*ptr;
printf("A=0x%016lX in memory\n", myvalue);

ptr = (uint64_t*)&b;
myvalue=*ptr;
printf("B=0x%016lX in memory\n", myvalue);
```

Floating-Point Errors

- **Contents of memory for floating-point variables *a* and *b* (64-bits = 16 hex digits):**

```
A=0x3FDCCCCCCCCCCCCC in memory
B=0x3FDCCCCCCCCCCCCD in memory
```

Foiled by the smallest of errors!

Remember:
C = 1100
D = 1101

Floating-Point Errors

- To test a floating point value for equality to some other number, it is best to declare a “nearness to x” epsilon value
- Example: instead of checking to see if floating point x is equal to 2 as follows:
  - if (x = 2) then ...
- Do this instead:
  - if (abs(x - 2) < epsilon) then ...
  - Must define epsilon to be small, but not too small!

Revised demo program:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <math.h>

int main()
{
  printf("Floating-Point Demo Program 2\n");
  double a, b;

  a = (58.0/40.0-1.0);
  b = (18.0/40.0);

  double epsilon = 1.0*pow(10,-9);
  if(abs(a-b)<epsilon)
    printf("A equals B!\n");
  else
    printf("A does not equal B!\n");

  return;
}
```

Actual output on test Linux machine:

```
Floating-Point Demo
Program 2
A equals B!
```

Floating-Point Errors

- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive
  - (a + b) + c = a + (b + c) or
  - a*(b + c) = ab + ac

May not be true!!

Floating-Point Errors

- Floating-point overflow and underflow can cause programs to crash
- **Overflow** occurs when there is no room to store the high-order bits resulting from a calculation
- **Underflow** occurs when a value is too small to store, possibly resulting in division by zero

Data Types

- **Where do I see all these data types in C/C++ programming?**
- `int` – Two's Complement number
- `unsigned int` – Plain old binary number
- `float` – IEEE single precision floating-point
- `double` – IEEE double precision floating-point