Floating point numbers
Floating point representation

- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$

- In binary (.=binary point)
  - $\pm1.xxxxxxxxx_2 \times 2^{yyyy}$

- Types float and double in C
floating-point IEEE standard 754-1985

Developed in response to divergence of representations

Now universally adopted

Two representations:

- single precision (32-bit)
- double precision (64-bit)

<table>
<thead>
<tr>
<th>Level</th>
<th>Width</th>
<th>Range at full precision</th>
<th>Precision[a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision</td>
<td>32 bits</td>
<td>$\pm1.18 \times 10^{-38}$ to $\pm3.4 \times 10^{38}$</td>
<td>Approximately 7 decimal digits</td>
</tr>
<tr>
<td>Double precision</td>
<td>64 bits</td>
<td>$\pm2.23 \times 10^{-308}$ to $\pm1.80 \times 10^{308}$</td>
<td>Approximately 16 decimal digits</td>
</tr>
</tbody>
</table>
- Most significant bit is the sign bit (0=positive, 1=negative)
- Fraction represents binary fractional part of the number, after being normalized
- The "1" before this fractional part is not stored, it is assumed.
- Exponent is biased to force negative/positive exponents sort in correct order:
  - 127 for single precision
  - 1203 for double precision

\[
x = (-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent-Bias})}
\]
reconstructing a floating-point number

Assume that the following is stored in memory: \[11000000101000...00\]

Break it down:

- sign = 1
- exponent = 129 - 127 = 2
- number = 1.01

Put it together:

\[-1.01 \times 2^2 \Rightarrow -101.0 \Rightarrow -5\] in decimal
storing a floating point number

Represent -0.75 in single-precision.

-0.75 decimal = -0.11 binary

normalize: -1.1 x 2^-1

sign = 1

exponent = -1+127 = 126 = 01111110 in binary

put it together: 101111110100..00 = 0xbf400000
convert to sp: +14.75

Steps:

1. determine sign bit
2. convert whole number part to binary
3. convert fraction part to binary
4. put 2 and 3 together
5. normalize 1… x 2^n
6. exponent = bias +n
7. convert biased exponent to binary
8. get fraction from step 5
Converting a base-10 decimal to binary

Convert whole-numbers by repeated division by 2

Convert fractional part by repeated multiplication by 2

Example: 0.75

1. multiply decimal portion by 2
2. keep the whole number part of the product
3. repeat until fraction is 0 or max digits
.085 to binary
stopped at max # digits

<table>
<thead>
<tr>
<th>fraction</th>
<th>x2</th>
<th>whole portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.085</td>
<td>0.17</td>
<td>0</td>
</tr>
<tr>
<td>0.17</td>
<td>0.34</td>
<td>0</td>
</tr>
<tr>
<td>0.34</td>
<td>0.68</td>
<td>0</td>
</tr>
<tr>
<td>0.68</td>
<td>1.36</td>
<td>1</td>
</tr>
<tr>
<td>0.36</td>
<td>0.72</td>
<td>0</td>
</tr>
<tr>
<td>0.72</td>
<td>1.44</td>
<td>1</td>
</tr>
<tr>
<td>0.44</td>
<td>0.88</td>
<td>0</td>
</tr>
<tr>
<td>0.88</td>
<td>1.76</td>
<td>1</td>
</tr>
<tr>
<td>0.76</td>
<td>1.52</td>
<td>1</td>
</tr>
<tr>
<td>0.52</td>
<td>1.04</td>
<td>1</td>
</tr>
<tr>
<td>0.04</td>
<td>0.08</td>
<td>0</td>
</tr>
<tr>
<td>0.08</td>
<td>0.16</td>
<td>0</td>
</tr>
<tr>
<td>0.16</td>
<td>0.32</td>
<td>0</td>
</tr>
<tr>
<td>0.32</td>
<td>0.64</td>
<td>0</td>
</tr>
<tr>
<td>0.64</td>
<td>1.28</td>
<td>1</td>
</tr>
<tr>
<td>0.28</td>
<td>0.56</td>
<td>0</td>
</tr>
<tr>
<td>0.56</td>
<td>1.12</td>
<td>1</td>
</tr>
<tr>
<td>0.12</td>
<td>0.24</td>
<td>0</td>
</tr>
<tr>
<td>0.24</td>
<td>0.48</td>
<td>0</td>
</tr>
<tr>
<td>0.48</td>
<td>0.96</td>
<td>0</td>
</tr>
<tr>
<td>0.96</td>
<td>1.92</td>
<td>1</td>
</tr>
<tr>
<td>0.92</td>
<td>1.84</td>
<td>1</td>
</tr>
<tr>
<td>0.84</td>
<td>1.68</td>
<td>1</td>
</tr>
<tr>
<td>0.68</td>
<td>1.36</td>
<td>1</td>
</tr>
<tr>
<td>0.36</td>
<td>0.72</td>
<td>0</td>
</tr>
<tr>
<td>0.72</td>
<td>1.44</td>
<td>1</td>
</tr>
<tr>
<td>0.44</td>
<td>0.88</td>
<td>0</td>
</tr>
<tr>
<td>0.88</td>
<td>1.76</td>
<td>1</td>
</tr>
<tr>
<td>0.76</td>
<td>1.52</td>
<td>1</td>
</tr>
</tbody>
</table>
Practice: Represent 17.75 in IEEE 754 SP

Check with this site: https://www.h-schmidt.net/FloatConverter/IEEE754.html
Practice

Reconstruct the base 10 number from the hex representation: \( 0x42085000 \)
Questions

1. What is the advantage of the order S-exp-frac?

2. The exponent being 8 v 11 bits affects:
   a. range of numbers
   b. precision of numbers

3. The fraction part being 23 v 52 bits affects:
   a. range of numbers
   b. precision of numbers
More about floating points

We can still have overflow.

- overflow happens when the exponent is too large for the exponent field
- underflow happens when a negative exponent is too large

Having double-precision helps. The range is:

- single precision: \( 2.0 \times 10^{-38} \) to \( 2.0 \times 10^{+38} \)
- double precision: \( 2.0 \times 10^{-308} \) to \( 2.0 \times 10^{+308} \)

However the primary advantage of double precision is greater accuracy.
IEEE 754 encoding

<table>
<thead>
<tr>
<th>Single precision</th>
<th>Double precision</th>
<th>Object represented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponent</strong></td>
<td><strong>Fraction</strong></td>
<td><strong>Exponent</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Nonzero</td>
<td>0</td>
</tr>
<tr>
<td>1–254</td>
<td>Anything</td>
<td>1–2046</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>2047</td>
</tr>
<tr>
<td>255</td>
<td>Nonzero</td>
<td>2047</td>
</tr>
</tbody>
</table>
Rounding errors

Not every number can be represented exactly, ex: 0.1

"If I had a dime for every time I've seen someone use FLOAT to store currency, I'd have $999.997634" -- Bill Karwin.
FP accuracy

32 bits gives us $2^{32}$, about 4 billion, unique bit patterns, but there are an infinite number of reals.

The IEEE 754 standard does not guarantee that every number can be represented, but that every machine using the standard will get the same results.
Rounding

IEEE 754 specifies two bits that are kept to the right during arithmetic operations. These bits are in the circuitry but not in the final result.

- the guard bit is the first extra bit to the right
- the round bit is the second bit to the right

The goal is to find the closest floating-point number that will fit into the format.

Further, a 'sticky' bit is set whenever there are nonzero bits to the right of the round bit. This is used in rounding.
Extra bits

These extra bits are in circuitry, not in the 32-bit or 64-bit representation.
Number representation

During the Gulf War in 1991, a US Patriot missile failed to intercept an Iraqi scud missile, resulting in 28 Americans being killed.

Cause: software updated a counter every 0.10 seconds, then multiplied the counter by 0.1 to compute the actual time.

Over 100 hours, the time was off by 0.34 seconds, enough for a scud to travel 500 meters.
extreme errors

problems occur if one argument is much smaller than the other since we need to match the exponents to add

\[(1.5 \times 10^{38}) + (1.0 \times 10^{0}) = 1.5 \times 10^{38}\]

The \(1.0 \times 10^{0}\) gets rounded out of existence
associativity break down

```c
#include <stdio.h>
int main (void)
{
    float x = 1.5e38;
    float y = -1.5e38;
    printf("%f\n", (x + y) + 1.0);
    printf("%f\n", x + (y + 1.0));
    return 0;
}
```

Output:
```
1 1.000000
2 0.000000
```
Questions

1. What do overflow/underflow mean in floating-point numbers?
2. What is NaN?
3. What is a denormalized number:
   a. the fraction part of the number cannot be represented in the number of bits
   b. the exponent part of the number cannot be represented in the number of bits
4. Denormalized numbers occur:
   a. near zero
   b. near the extremes +/- of magnitude of numbers that can be represented
5. True or false. Arithmetic associativity can break down when adding numbers at opposite extremes (most large and most small)
MIPS FP registers

Click on Coprocessor 1 to see them

Coprocessor 1 is a simulated floating-point coprocessor

The fp registers can be accessed as single-precision (32-bits) or double (64-bits)

Even registers can hold 64 bits
## ARITHMETIC CORE INSTRUCTION SET

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FORMAT</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch On FP True</td>
<td>bcei</td>
<td>if(FPcond)PC=PC+4+BranchAddr</td>
</tr>
<tr>
<td>Branch On FP False</td>
<td>bcef</td>
<td>if(!FPcond)PC=PC+4+BranchAddr</td>
</tr>
<tr>
<td>Divide</td>
<td>div</td>
<td>R=Rs/Rt; Hi=Rs%Rt</td>
</tr>
<tr>
<td>Divide Unsigned</td>
<td>divu</td>
<td>R=Rs/Rt; Hi=Rs%Rt</td>
</tr>
<tr>
<td>FP Add Double</td>
<td>add.d</td>
<td>F[fd],F[fd+1] = (F[fs],F[fs+1]) + (F[ft],F[ft+1])</td>
</tr>
<tr>
<td>FP Compare Single</td>
<td>c.rs*</td>
<td>FPcond = (F[fs] op F[ft]) ( ? : 0 )</td>
</tr>
<tr>
<td>FP Compare Double</td>
<td>c.rs*</td>
<td>FPcond = (F[fs],F[fs+1]) op (F[ft],F[ft+1]) ( ? : 0 )</td>
</tr>
<tr>
<td>FP Divide Double</td>
<td>div.d</td>
<td>F[fd],F[fd+1] = (F[fs],F[fs+1]) / (F[ft],F[ft+1])</td>
</tr>
<tr>
<td>FP Multiply Double</td>
<td>mul.d</td>
<td>F[fd],F[fd+1] = (F[fs],F[fs+1]) * (F[ft],F[ft+1])</td>
</tr>
<tr>
<td>FP Subtract Single</td>
<td>sub.s</td>
<td>F[fd] = F[fs] - F[ft]</td>
</tr>
<tr>
<td>FP Subtract Double</td>
<td>sub.d</td>
<td>F[fd],F[fd+1] = (F[fs],F[fs+1]) - (F[ft],F[ft+1])</td>
</tr>
<tr>
<td>Load FP Single</td>
<td>lwcl</td>
<td>F[rt] = M[R[rs]+SignExtImm]</td>
</tr>
<tr>
<td>Load FP Double</td>
<td>ldcl</td>
<td>F[rt] = M[R[rs]+SignExtImm]</td>
</tr>
<tr>
<td>Move From Hi</td>
<td>mfhq</td>
<td>R[rd] = Hi</td>
</tr>
<tr>
<td>Move From Lo</td>
<td>mflo</td>
<td>R[rd] = Lo</td>
</tr>
<tr>
<td>Move From Control</td>
<td>mfco</td>
<td>R[rd] = CR[rs]</td>
</tr>
<tr>
<td>Multiply</td>
<td>mult</td>
<td>R[Hi,Lo] = R[rs] * R[rt]</td>
</tr>
<tr>
<td>Multiply Unsigned</td>
<td>multu</td>
<td>R[Hi,Lo] = R[rs] * R[rt]</td>
</tr>
<tr>
<td>Store FP Single</td>
<td>swcl</td>
<td>M[R[rs]+SignExtImm] = F[rt]</td>
</tr>
<tr>
<td>Store FP Double</td>
<td>addcl</td>
<td>M[R[rs]+SignExtImm] = F[rt]</td>
</tr>
</tbody>
</table>

Opcodes:
- FMT: FT
- FUNCT: (Hex)
FP arithmetic instructions

Replace .x with .s (single precision) or .d (double precision)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.x FPdest, FPsrc1, FPsrc2</td>
<td>FPdest = FPsrc1 + FPsrc2</td>
</tr>
<tr>
<td>sub.x FPdest, FPsrc1, FPsrc2</td>
<td>FPdest = FPsrc1 - FPsrc2</td>
</tr>
<tr>
<td>mul.x FPdest, FPsrc1, FPsrc2</td>
<td>FPdest = FPsrc1 * FPsrc2</td>
</tr>
<tr>
<td>div.x FPdest, FPsrc1, FPsrc2</td>
<td>FPdest = FPsrc1 \ FPsrc2</td>
</tr>
<tr>
<td>abs.x FPdest, FPsrc</td>
<td>FPdest = abs(FPsrc)</td>
</tr>
<tr>
<td>neg.x FPdest, FPsrc</td>
<td>FPdest = negate(FPsrc)</td>
</tr>
</tbody>
</table>
### load and store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwcl FPdest, address</td>
<td>FPdest = (address)</td>
</tr>
<tr>
<td>swcl FPsrc, address</td>
<td>(address) = FPsrc</td>
</tr>
<tr>
<td>Idc1 FPdest, address</td>
<td>FPdest = (address)</td>
</tr>
<tr>
<td>sdcl FPsrc, address</td>
<td>(address) = FPsrc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.x FPdest, address</td>
<td>FPdest = (address)</td>
</tr>
<tr>
<td>s.x FPsrc, address</td>
<td>(address) = FPsrc</td>
</tr>
</tbody>
</table>
FP move instructions

Move between coprocessor 1 registers and the general-purpose registers

mov.x can be mov.s or mov.d
FP conversion

Replace .x with .s or .d

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvt.x.w FPdest, FPsrc</td>
<td>FPdest = to_FP(FPsrc integer)</td>
</tr>
<tr>
<td>cvt.w.x FPdest, FPsrc</td>
<td>FPdest = to_int(FPsrc float)</td>
</tr>
<tr>
<td>cvt.d.s FPdest, FPsrc</td>
<td>FPdest = to_double(FPsrc single-precision)</td>
</tr>
<tr>
<td>cvt.s.d FPdest, FPsrc</td>
<td>FPdest = to_single(FPsrc double)</td>
</tr>
</tbody>
</table>
FP compare and branch

Replace .x with .s or .d

c is the floating point condition flag

"c1" for coprocessor 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.eq.x FPsrc1, FPsrc2</td>
<td>c=1 if FPsrc1 == FPsrc2</td>
</tr>
<tr>
<td>c.le.x FPsrc1, FPsrc2</td>
<td>c=1 if FPsrc1 &lt;= FPsrc2</td>
</tr>
<tr>
<td>c.lt.x FPsrc1, FPsrc2</td>
<td>c=1 if FPsrc1 &lt; FPsrc2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bctl label</td>
<td>branch if c=1 (true)</td>
</tr>
<tr>
<td>bclf label</td>
<td>branch if c=0 (false)</td>
</tr>
</tbody>
</table>
FP example: area of a circle

```c
#include <stdio.h>

int main(void) {
    // area = pi * r * r
    double pi = 3.1415926535897924;
    double r = 12.345678901234567;
    double area;

    area = pi * r * r;
    printf("%f", area);
    return 0;
}
```

```assembly
# FP example to compute the area of a circle
.data
    pi: .double 3.1415926535897924f
    rad: .double 12.345678901234567f
.text
main:
    l.d $f0, pi       # $f0 = pi
    l.d $f4, rad     # $f4 = radius
    mul.d $f12, $f4, $f4 # $f12 = rad^2
    mul.d $f12, $f12, $f0 # $f12 = rad^2 * pi
    li $v0, 3         # output answer
    syscall
.exit:
    li $v0, 10       # terminate program
    syscall
```
FP example: fahrenheit to celsius

```c
#include <stdio.h>

int main(void) {
    // C = 5/9 * (F - 32)
    float fahr = 72;
    float celsius;
    celsius = 5.0/9.0 * (fahr - 32);
    printf("%f", celsius);
    return 0;
}
```
Summary

- floating point registers can be stored as single precision or double precision
- double precision FP registers have even numbers
- arithmetic is of form: add.x where you replace x with s or d for single or double precision
- special instructions allow you to move registers to and from coprocessor 1 to the main coprocessor, and load/store to memory
- other instructions let you convert from integer to floating point and back
- we have to use special compare and branch instructions for floating point registers
Practice

Given the following in .data:

```
.data
x:   .float  3.8
y:   .float  4.2
```

Write code to calculate the average of x and y and output it.
Find the errors in this code

4 lines with errors

- 3 assemble errors
- 1 run time error

```
.data
x: .float 3.8
y: .float 4.2

.text
lw $f2, x
lw $f3, y
add.s $f0, $f2, $f3
li $t1, 2.0
mtc1 $t1, $f1
cvt.w.s $f1, $f1
div.s $f12, $f0, $f1
li $v0, 2
syscall
```
Debugging

If use "cvt.w.s" instead of "cvt.s.w" the answer is infinity.

Look at register values and use the Schmidt site.

Answer $f_{12}=\infty = 0\ 11111111\ 0000...00$

Sum in $f_0 = 8$ is ok

2.0 in $f_1$ is 00000000 that where the error is!
Practice

Create a BMI Calculator

```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int height = 0, weight = 0;
    double bmi;
    string name;

    // Prompt user for their data
    cout << "What is your name? ";
    cin >> name;

    cout << "Please enter your height in inches: ";
    cin >> height;
    cout << "Now enter your weight in pounds (round to a whole number): ";
    cin >> weight;

    // Calculate the bmi
    weight *= 703;
    height *= height;
    bmi = static_cast<double>(weight) / height;

    // Output the results
    cout << name << "", your bmi is: " << bmi << endl;

    if (bmi < 18.5)
        cout << "This is considered underweight. \n";
    else if (bmi < 25)
        cout << "This is a normal weight. \n";
    else if (bmi < 30)
        cout << "This is considered overweight. \n";
    else
        cout << "This is considered obese. \n";

    return 0;
}
```