Floating point numbers

Floating point representation

- Like scientific notation
  - -2.34×10^{56} normalized
  - $+0.002 \times 10^{-4}$ not normalized
  - $+987.02 \times 10^{9}$ not normalized
- In binary (.=binary point)
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

floating-point IEEE standard 754-1985

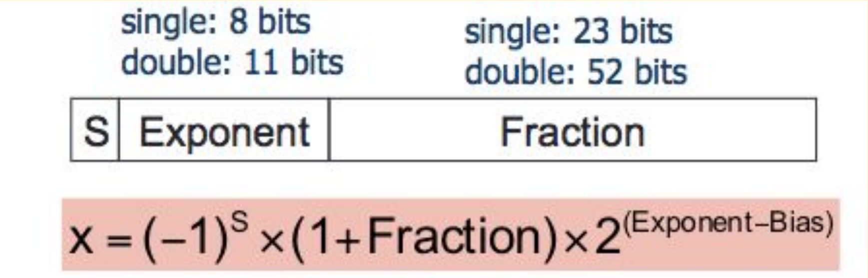
Developed in response to divergence of representations

Now universally adopted

Two representations:

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

Level	Width	Range at full precision	Precision[a]
Single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	Approximately 7 decimal digits
Double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	Approximately 16 decimal digits



- 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

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 x 2^2 == -101.0 == -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

1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S	exponent								fraction																						
0	1	0	0	0	0	0	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

fraction	x2	whole portion
0.75	1.5	1
0.5	1	1
0	0	0

.085 to binary

stopped at max # digits

fraction	x2	whole portior
0.085	0.17	0
0.17	0.34	0
0.34	0.68	0
0.68	1.36	1
0.36	0.72	0
0.72	1.44	1
0.44	0.88	0
0.88	1.76	1
0.76	1.52	1
0.52	1.04	1
0.04	0.08	0
0.08	0.16	0
0.16	0.32	0
0.32	0.64	0
0.64	1.28	1
0.28	0.56	0
0.56	1.12	1
0.12	0.24	0
0.24	0.48	0
0.48	0.96	0
0.96	1.92	1
0.92	1.84	1
0.84	1.68	1
0.68	1.36	1
0.36	0.72	0
0.72	1.44	1
0.44	0.88	0
0.88	1.76	1
0.76	1.52	1

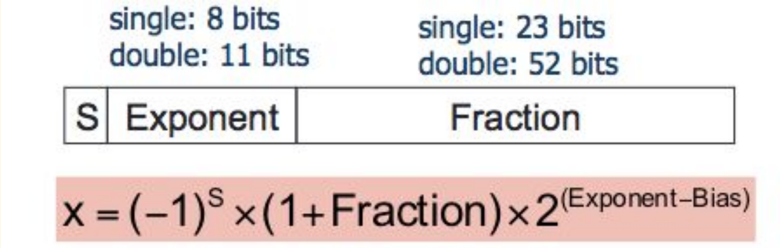
Practice: Represent 17.75 in IEEE 754 SP

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: almost 2.0×10^{-38} to $2.0 \times 10^{+38}$
- double precision: range is almost: 2.0×10^{-308} to $2.0 \times 10^{+308}$

However the primary advantage of double precision is greater accuracy.

IEEE 754 encoding

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Rounding errors

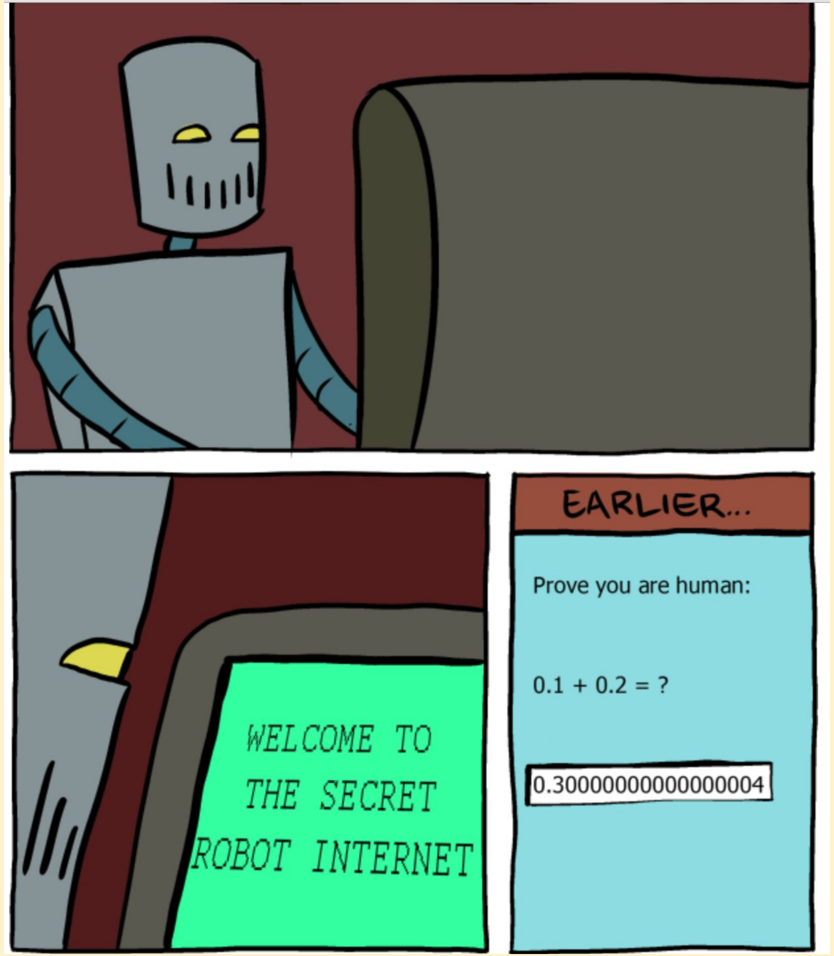
Not every number can be represented exactly, ex: 0.1

"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.

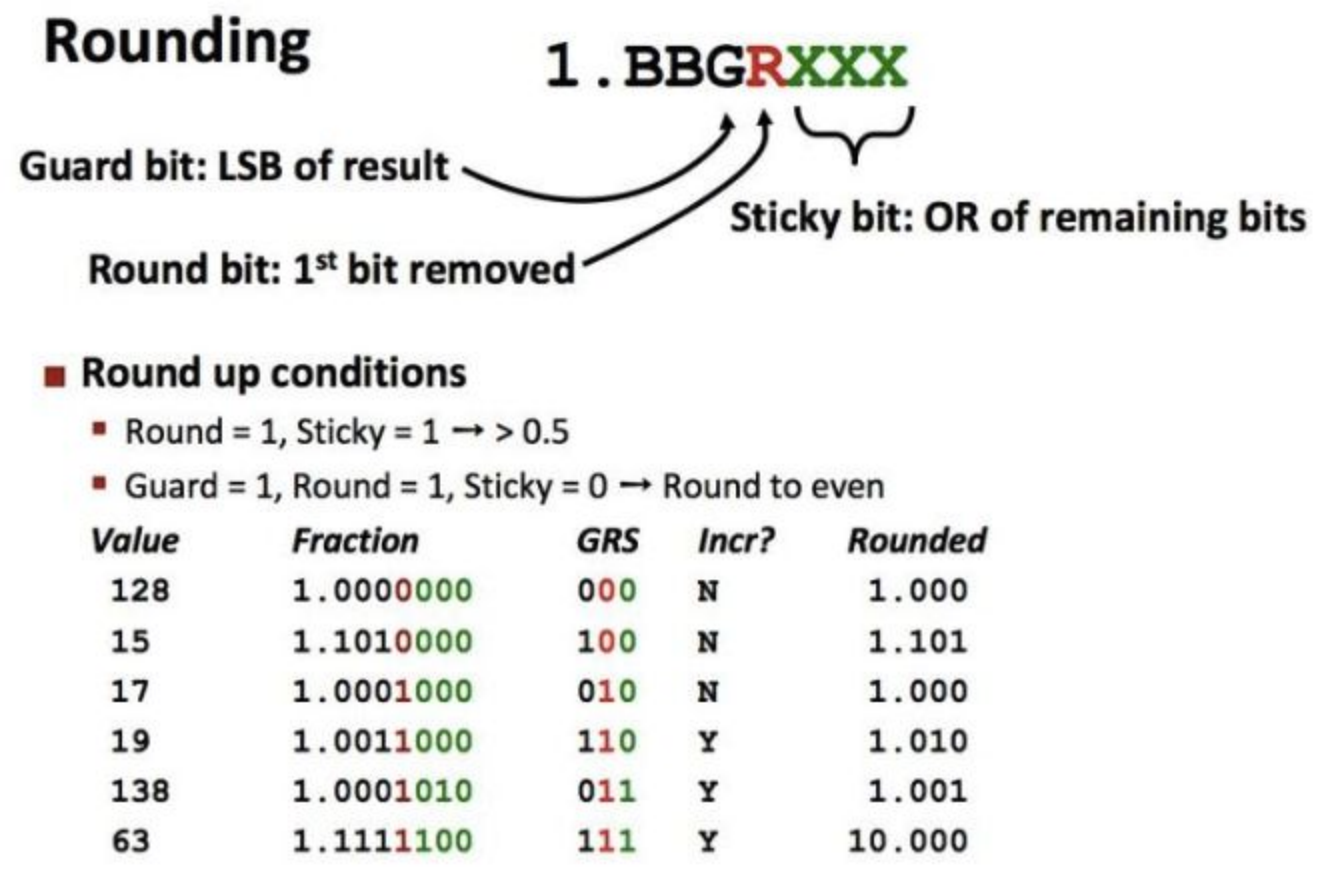
- the guard bit is the first extra bit to the right
- the round bit is the second bit to the right

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.



Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round up conditions**
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

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 1.0×10^{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:

```
1.000000
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

Name	Float	Double
$f0	0x00000000	0x0000000000000000
$f1	0x00000000	
$f2	0x00000000	0x0000000000000000
$f3	0x00000000	
$f4	0x00000000	0x0000000000000000
$f5	0x00000000	
$f6	0x00000000	0x0000000000000000
$f7	0x00000000	
$f8	0x00000000	0x0000000000000000
$f9	0x00000000	
$f10	0x00000000	0x0000000000000000
$f11	0x00000000	
$f12	0x00000000	0x0000000000000000
$f13	0x00000000	
$f14	0x00000000	0x0000000000000000
$f15	0x00000000	
$f16	0x00000000	0x0000000000000000
$f17	0x00000000	
$f18	0x00000000	0x0000000000000000
$f19	0x00000000	
$f20	0x00000000	0x0000000000000000
$f21	0x00000000	
$f22	0x00000000	0x0000000000000000
$f23	0x00000000	
$f24	0x00000000	0x0000000000000000
$f25	0x00000000	
$f26	0x00000000	0x0000000000000000
$f27	0x00000000	
$f28	0x00000000	0x0000000000000000
$f29	0x00000000	
$f30	0x00000000	0x0000000000000000

MIPS card

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION	OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr (4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt] (6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]	11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0	11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0	11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)				
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]	11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]	11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}	11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]	11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}	11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm] (2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; (2) F[rt+1]=M[R[rs]+SignExtImm+4]	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi	0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo	0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]	10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]	0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt] (6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt	0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt] (2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; (2) M[R[rs]+SignExtImm+4] = F[rt+1]	3d/--/--/--

FP arithmetic instructions

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

Instruction	Action
add.x FPdest, FPsrc1, FPsrc2	FPdest = FPsrc1 + FPsrc2
sub.x FPdest, FPsrc1, FPsrc2	FPdest = FPsrc1 – FPsrc2
mul.x FPdest, FPsrc1, FPsrc2	FPdest = FPsrc1 * FPsrc2
div.x FPdest, FPsrc1, FPsrc2	FPdest = FPsrc1 \ FPsrc2
abs.x FPdest, FPsrc	FPdest = abs(FPsrc)
neg.x FPdest, FPsrc	FPdest = negate(FPsrc)

load and store

Instruction	Action
lwc1 FPdest, address	FPdest = (address)
swc1 FPsrc, address	(address) = FPsrc
ldc1 FPdest, address	FPdest = (address)
sdc1 FPsrc, address	(address) = FPsrc

Pseudo-instruction	Action
l.x FPdest, address	FPdest = (address)
s.x FPsrc, address	(address) = FPsrc

FP move instructions

Move between coprocessor 1 registers and the general-purpose registers

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

Instruction	Action
mfc1 Rdest, FPsrc	Rdest = FPsrc
mtc1 Rsrc, FPdest	FPdest = Rsrc
mov.x FPdest, FPsrc	FPdest = FPsrc

FP conversion

Replace .x with .s or .d

Instruction	Action
cvt.x.w FPdest, FPsrc	FPdest = to_FP(FPsrc integer)
cvt.w.x FPdest, FPsrc	FPdest = to_int(FPsrc float)
cvt.d.s FPdest, FPsrc	FPdest = to_double(FPsrc single-precision)
cvt.s.d FPdest, FPsrc	FPdest = to_single(FPsrc double)

FP compare and branch

Replace .x with .s or .d

c is the floating point condition flag

"c1" for coprocessor 1

Instruction	Action
c.eq.x FPsrc1, FPsrc2	c=1 if FPsrc1 == FPsrc2
c.le.x FPsrc1, FPsrc2	c=1 if FPsrc1 <= FPsrc2
c.lt.x FPsrc1, FPsrc2	c=1 if FPsrc1 < FPsrc2

Instruction	Action
bclt label	branch if c=1 (true)
bclf label	branch if c=0 (false)

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;
}
```

```
# FP example to compute the area of a circle
        .data
pi:     .double         3.1415926535897924
rad:    .double         12.345678901234567
        .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;
}
```

```
# f2c1.asm
# program converts Fahrenheit temperature to Celsius
# C = (F - 32) * 5/9
        .data
const5: .float  5.0
const9: .float  9.0
const32:.float  32
fahr:   .float  72.0
celc:   .float  0
msgf:   .asciiz "\nFahrenheit temperature of "
msgc:   .asciiz " is equivalent to Celsius temp "
        .text
main:
        lwc1    $f12, fahr
        #lwc1   $f16, const5
        # or use these 3 instrucions:
        li      $t0, 5
        mtc1    $t0, $f16
        cvt.s.w $f16, $f16
        #
        # can't do this:
        #li     $f16, 5
        #cvt.s.w        $f16, $f16
        #
        lwc1    $f18, const9
        div.s   $f16, $f16, $f18    # $f16 = 5 / 9
        lwc1    $f18, const32
        sub.s   $f18, $f12, $f18    # $f18 = F - 32
        mul.s   $f0, $f16, $f18     # $f0 = (F - 32) * 5/9
        swc1    $f0, celc

        # display results
        li      $v0, 4  # print msgf
        la      $a0, msgf
        syscall
        li      $v0, 2
        lwc1    $f12, fahr
        syscall
        li      $v0, 4  # print msgc
        la      $a0, msgc
        syscall
        li      $v0, 2
        lwc1    $f12, celc
        syscall

        # exit program
exit:   li      $v0, 10
        syscall
```

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 $f12=infinity = 0 11111111 0000...00

Sum in $f0 = 8 is ok

2.0 in $f1 is 00000000 that where the error is!

Name	Float
$f0	0x41000000
$f1	0x00000000
$f2	0x40733333
$f3	0x40866666
$f4	0x00000000
$f5	0x00000000
$f6	0x00000000
$f7	0x00000000
$f8	0x00000000
$f9	0x00000000
$f10	0x00000000
$f11	0x00000000
$f12	0x7f800000

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;
}

```