Lecture 5: Floating Point Math

ME/AE 6705

Introduction to Mechatronics

Dr. Jonathan Rogers

Lesson Objectives

- Understand how decimal numbers (floating point numbers) are represented on microprocessors
- Be able to convert a decimal number into floating point representation
- Understand the limits of precision using floating point numbers
- Understand fixed point representations
- Understand methods of explicit type conversion (casting) in C programming

Limits of Integer Math

- Last class we learned how integer arithmetic is performed on microprocessors
- In many applications, integer math is not sufficient for the calculations we need

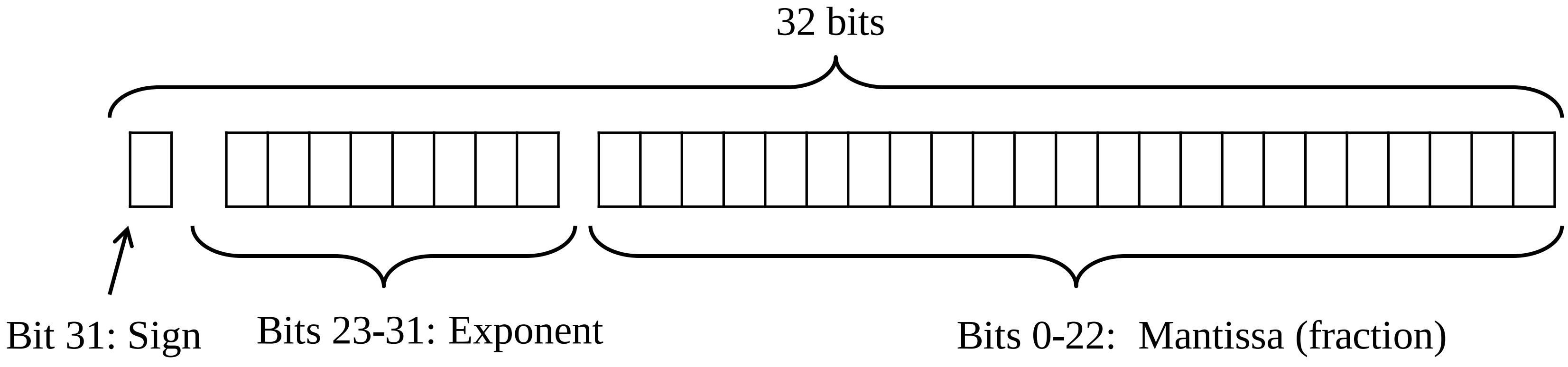
$$\frac{1024}{1099} = 0.9318 \quad \longleftarrow$$

Zero is often not an acceptable result

  – Example: Robot determines its speed = 4 ft/s and needs to travel 3 ft. Does it have 0 sec or 0.75 sec left to travel?

Floating Point Numbers

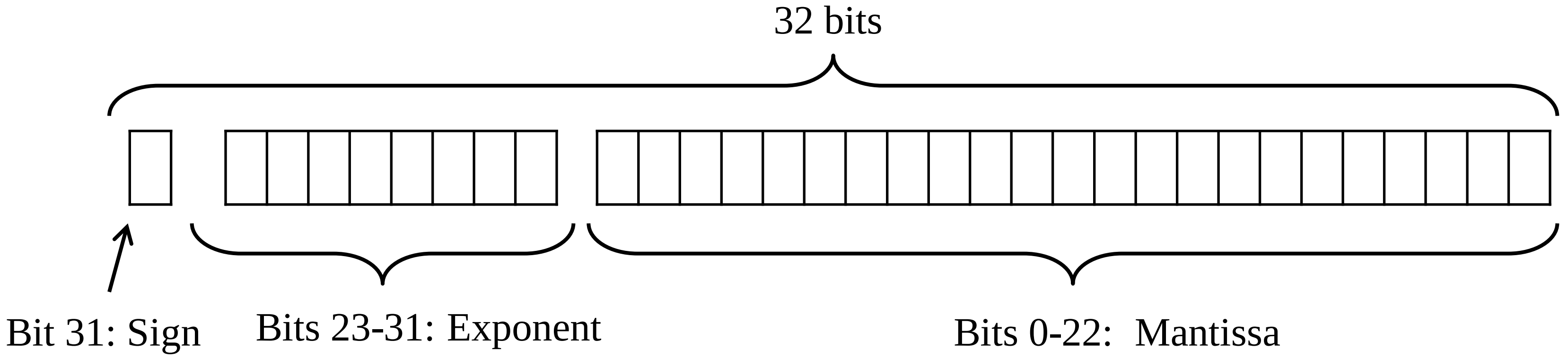
- Floating point numbers are a way to represent decimal numbers with binary digits
- Floating point numbers <u>almost always</u> use 32-bit or 64-bit representation
  - 64-bit can represent larger range of decimal numbers to higher precision
- IEEE 754 floating point standard (32-bit):



Floating Point Numbers

- Decimal number *A* represented in floating point as follows:

$$A = (\text{Sign}) \times 2^{\text{exponent}} \times \text{mantissa}$$



Floating Point Numbers

- Sign bit – 1 if negative number, 0 if positive number
- Mantissa – represents number between 1 and 2
  - Mantissa computed by letting each bit represent negative power of 2 in decreasing order

$$\text{Mantissa} = 1 + (\text{bit } 22) \times 2^{-1} + (\text{bit } 21) \times 2^{-2} + \ldots (\text{bit } 0) \times 2^{-23}$$

$$= 1 + (\text{bit } 22) \times \frac{1}{2} + (\text{bit } 21) \times \frac{1}{4} + \ldots (\text{bit } 0) \times \frac{1}{8388608}$$

$$= 1 + \sum_{i=0}^{22} (\text{bit } i) \times 2^{i-23}$$

Note: Mantissa value is always between 1 and 2. It is 1 when all mantissa bits are zero, it is approximately 2 when all mantissa bits are 1.

Floating Point Numbers

- Exponent represented by bits 23-30 (8 bits total)
  - Exponent is binary integer represented by 8 bits minus 127
  - The -127 is used so both positive and negative exponents can be represented
  - Value of exponent runs between 128 and -127

$$\text{Exponent} = \left(\sum_{i=23}^{30} (\text{bit}\, i) \times 2^{i-23} \right) - 127$$

$$A = (\text{Sign}) \times 2^{\text{Exponent}} \times \text{mantissa}$$

Used here in calculation of floating point number

Floating Point Number Example

- Determine decimal value of following floating point number:

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

- Step 1: Number is positive since sign bit is 0.
- Step 2: Compute exponent value
  - Exponent = 2^7-127 = 128-127 = 1
- Step 3: Compute mantissa
  - Mantissa = 1 + 1/2 + 1/4 + 1/8 + 1/16 = 1.9375

Floating Point Number Example

- Putting it all together:

$$A = (\text{Sign}) \times 2^{\text{Exponent}} \times \text{mantissa}$$

$$= 1 \times 2^1 \times 1.9375 = \mathbf{3.875}$$

- Notes:
  - Range of numbers that 32-bit floating point can represent is +/-1.18×10^{-38} to +/-3.4×10^{38}
  - Zero is represented by mantissa = 0 and exponent = 0 (special case)

Floating Point Number Example

- Example problem: Determine the decimal number represented by:

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

Floating Point Numbers

- Converting decimal number *A* to floating point representation is a bit trickier
  - Step 1: Note whether number is + or – to determine sign bit
  - Step 2: Let $2^z = |A|$ and then determine exponent *e* as *e* = floor(*z*)
  - Step 3: Determine mantissa by recursively adding negative powers of 2

Floating Point Number Example

- Example problem: Convert 0.15625 into floating point representation.
  - Step 1: Sign bit will be 0 since number is positive
  - Step 2: Find the exponent

$$2^z = 0.15625$$

$$z = \log_2\left(0.15625\right) = \frac{\ln\left(0.15625\right)}{\ln\left(2\right)}$$

$$= -2.6780719$$

$$\Rightarrow \quad e = \text{floor}(z) = -3$$

Georgia Tech

Floating Point Number Example

- Example continued…
  - Step 3: Computing the mantissa

$$0.15625 = (1)\,(2^{-3})\,x$$ $\longleftarrow$ (definition of floating point number, x is mantissa)

$$x = \frac{0.15625}{2^{-3}} = 1.25 \qquad (x \text{ will always be } \geq 1)$$

- *Start at k=0, compare x-1 to 2^{-k}, if $x \geq 2^{-k}$ then set k^{th} bit to* 1*, otherwise set it to* 0
- *Then set $x = x - 2^{-k}$ and repeat until k = 23 or x = 0*

$$k = 0 \qquad 1.25 - 1 \geq 2^{-1} \qquad \text{no} \rightarrow \text{set } k^{th} \text{ bit to } 0 \qquad x = 1.25 - 1 = 0.25$$

$$k = 1 \qquad 0.25 \geq 2^{-2} \qquad \text{yes} \rightarrow \text{set } k^{th} \text{ bit to } 1 \qquad x = 0.25 - 0.25 = 0$$

Floating Point Number Example

- Example continued…
- Putting everything together, 1.25 is represented in 32-bit floating point format as

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

Note: This is -3+127 = 124

  – Writing this in hexadecimal,

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

Floating Point Number Example

- Example continued…
- Putting everything together, 1.25 is represented in 32-bit floating point format as

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

  - Writing this in hexadecimal,

$$\underbrace{0\ 0\ 1\ 1\ 1\ 1\ 1\ 0}_{3E}\ \underbrace{0\ \ 0\ 1\ 0\ 0\ 0\ 0\ 0}_{20}\ \underbrace{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0}_{00}\ \underbrace{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0}_{00}$$

Floating Point Precision

- 32-bit floating point format can only represent a number exactly if it can be created by summing powers of 2 from 2^{-1} to 2^{-23}
  - i.e., using 24 bits
  - In previous example, our number 0.15625 could be represented exactly because $0.15625 = 2^{-3} + 2^{-5}$
- If the above is not true for the number we are trying to represent, the number will be approximated in floating point by truncating it at the 24th bit
  - This means it will be correct to the ~7th decimal place

Precision Example

- Example: How precisely is 16π represented in floating point format (32 bit)?

- Solution:
  - By repeating steps from previous example, we find that the floating point representation of 16π is

$$0\text{x}\overbrace{84}^{\text{sign bit + exponent}}\underbrace{490\text{FDB}}_{\text{mantissa}}$$

Precision Example

- The exponent value is 0x84 which is 132 in decimal, thus

$$\text{exponent} = 132 - 127 = 5$$

- The mantissa (or fraction) is 0x490FDB which in binary is

100	1001	0000	1111	1101	1011
4	9	0	F	D	B

Precision Example

- Writing the mantissa *f* in decimal,

$$f = 1 + 2^{-1} + 2^{-4} + 2^{-7} + 2^{-12} + 2^{-13} + 2^{-14} +$$
$$2^{-15} + 2^{-16} + 2^{-17} + 2^{-19} + 2^{-20} + 2^{-22} + 2^{-23}$$
$$= 1.5707963705$$

- Thus, the final decimal value of the floating point representation is:

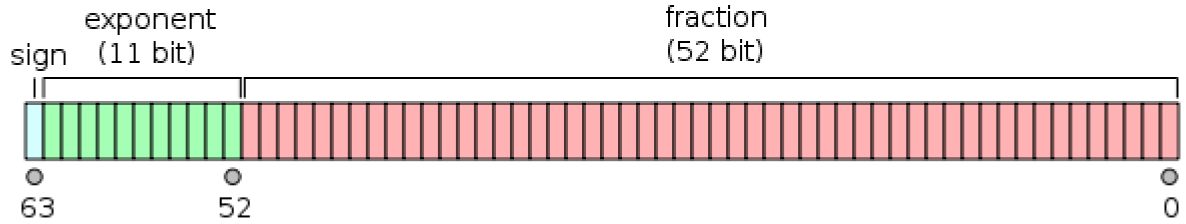
$$1 \times 2^{5} \times 1.5707963705 = 50.265483856$$

Actual value of 16π = 50.265482457

Correct to ~6th decimal place

Floating Point Precision

- So far, we have discussed 32-bit floating point numbers
  - In C, these are numbers of type "float"
- In addition, 64-bit floating point numbers are defined to give higher precision
  - In C, these are numbers of type "double"
  - Uses 11-bit exponent instead of 8
  - Uses 52-bit mantissa (or fraction) instead of 23



Floating Point Precision

- Double precision (64-bit) floating point allows much greater precision and range compared to 32-bit

Type	Width	Range at Full Precision	Precision
Single precision (float)	32 bits	+/-10^{-38} to +/-10^{38}	Approximately 7 decimal digits
Double precision (double)	64 bits	+/-10^{-305} to +/-10^{305}	Approximately 15 decimal digits

Fixed Point Numbers

- Processing floating point numbers (adding, multiplying, dividing, etc) requires a special set of registers on the microprocessor
  - This is called a Floating Point Unit (FPU)
  - <u>Many microcontrollers do not have FPU's</u>
  - If FPU is not available or not activated, processor cannot handle floating point numbers
- Furthermore, floating point math is slower than integer math even if FPU is available
  - Thus, alternative called Fixed Point sometimes used

Fixed Point Numbers

- Fixed point numbers are used to represent non-integer values
- They consist of an <u>integer</u> *I* and a <u>fixed constant</u> Δ
  - Integer part *I* is stored in memory as normal integer type
  - Constant Δ is not stored in memory and cannot be changed, but rather is set by programmer when software is written

$$f = I \times \Delta$$

Note: Δ is usually specified in comments of the program.

Fixed Point Numbers

- Δ is a multiplier value that we use during input and output to convert physical units to integers which are stored in memory and manipulated
  - Example: Δ = 0.001. Thus, to represent 2.314 in memory, we would simply use an integer value of 2314.
  - Before inputting values to program, we would divide our values by 0.001.
  - After outputting values from program, we multiply integer values by 0.001.

Fixed Point Numbers

- Decimal fixed point number (Δ power of 10)
  - Easier for input and output

$$\text{Decimal fixed point number} = I \times 10^{m} \quad \text{for some fixed constant } m < 0$$

- Binary fixed point number (Δ power of 2)
  - Easier for mathematical calculations

$$\text{Binary fixed point number} = I \times 2^{n} \quad \text{for some fixed constant n} < 0$$

  - Note: Binary fixed point numbers are easier to use if you have different values of Δ throughout your program and you need to convert between them.

Fixed Point Example

- Example: Suppose for your program you decide to use $\Delta = 0.001$. How would 3π be represented in fixed point format in your program?
  - Furthermore, what size integer is required to represent it?

Fixed Point Example

- Example: Suppose for your program you decide to use $\Delta = 0.001$. How would 3π be represented in fixed point format in your program?
  - Furthermore, what size integer is required to represent it?
- Solution:

$$3\pi = 9.42477796076938$$

$$\rightarrow 9.42477796076938 / 0.001 = 9424.7779607...$$

  - Thus, an integer of 9425 would be used to represent 3π in memory
  - Note this integer must be at least 16 bits to avoid problems with overflow

Fixed Point Example

- Example: Suppose for your program you decide to use $\Delta = 2^{-8}$. How would 3π be represented in fixed point format in your program?
  - Furthermore, what size integer is required to represent it?

Fixed Point Example

- Example: Suppose for your program you decide to use $\Delta = 2^{-8}$. How would 3π be represented in fixed point format in your program?
  - Furthermore, what size integer is required to represent it?
- Solution:

$$3\pi / 2^{-8} = 9.4247779607 / 0.00390625$$
$$= 2412.74315795...$$

  - Thus, an integer of value 2413 would be used to represent 3π in memory
  - Note this integer must be at least 16 bits to avoid problems with overflow

Fixed Point Arithmetic

- Δ can be thought of as just a "scale factor" that converts units in your program to physical units
- For example, you may be dealing with Volts in the "physical" domain, but processor will deal with millivolts if $\Delta = 0.001$.
- As a result, fixed point numbers can be added, subtracted, multiplied, and divided exactly the same as with integers
  - If they have different values of Δ, conversion to same Δ must be done first

Georgia Tech

Fixed Point Example

- Suppose you have a value of B = 4386 in memory, which is a fixed point value with $\Delta = 0.001$ representing voltage measured across two pins
- You have another value of C = 357421 which is fixed point value with $\Delta = 10^{-6}$ representing voltage measured across two other pins
- You wish to add these voltages in your program so that D = B + C is a fixed point value. How do you do this so as to maintain <u>maximum precision</u> (minimum truncation error)?

Georgia Tech

Fixed vs Floating Point

Floating Point

- Uses more memory and is slower
  - All numbers use at least 32 bits
- Must have FPU available and active
- Can handle large variation in range of numbers
- Called “floating point” because number of decimal places varies depending on how large number is

Fixed Point

- Uses less memory and is faster
- Will always work regardless of processor (even if no FPU available)
- Only accurate if range of numbers is known and small
- Called “fixed point” because number of decimal places always fixed for each number

Type Conversions

- C99 compilers will automatically perform type conversions when arithmetic is done on two different variable types
- When two types are operated on, lower order is converted to higher with result stored as higher order
- Order is:
  - Double
  - Float
  - Signed 32 bit
  - Unsigned 32 bit
  - Signed 16 bit
  - Unsigned 16 bit
  - Signed 8 bit
  - Unsigned 8 bit

"promotion" ↑

"demotion" ↓

Type Conversions

- Consider fragment of C program below:

```
int8_t num1 = 77 ;
int8_t num2 = 82 ;
int16_t num3 = -534 ;
int32_t res = 0 ;

res = num3 - (num1 + num2) ;
```

Added as 8 bit, stored as 8 bit

8 bit converted to 16 bit,
added as 16 bit

16 bit converted to 32 bit

Type Conversions

- Explicit conversion from one variable to type to another can be done at any time by placing (type) directly before variable
  - This is called typecasting
- Consider previous case. Adding num1 and num2 would cause overflow (since 77 + 82 > 127)
- Can fix this by typecasting one number to 16-bit integer

```
int8_t num1 = 77 ;
int8_t num2 = 82 ;
int16_t num3 = -534 ;
int32_t res = 0 ;

res = (int32_t)num3 - ((int16_t)num1 + num2) ;
```

Type Conversions

- Similarly, typecasting is also done for 32 bit and 64 bit floating point types

```
float num1 = 77.6598 ;
float num2 = 82.4758 ;
int16_t num3 = -534 ;
double res = 0 ;

res = num3 - (num1 / num2) ;
```

  - num1 and num2 divided as 32 bit floats, num3 converted to 32 bit float and added to –(num1+num2)
  - Result then converted to 64 bit float and set equal to res

Type Conversions

- Using explicit typecasting is always better as it helps to keep track of precision during math operations

```
float num1 = 77.6598 ;
float num2 = 82.4758 ;
int16_t num3 = -534 ;
double res = 0 ;

res = (double)((float)num3 - (num1 / num2)) ;
```

printf Function in C

- In C, you can print variables to screen during runtime using **printf** function
  - Supported for TI microcontroller in debug mode
- Example functionality:

```
printf("Value of num1 is %u\n", num1);
```

Prints num1 to screen, then ends line (return).

```
printf("num2 = %f  num3 = %.10f\n",num2,num3);
```

Prints num2 and num3 to screen, then ends line (return).

Important: %XX characters tell compiler how to interpret value during printing process (i.e., variable type).

printf Function in C

Specifier	Output	Example
`%c`	Character	a
`%d` or `%i`	Signed decimal integer	-392
`%ld`	Signed 32 bit long decimal integer	65846214
`%e`	Scientific notation	6.022141e23
`%E`	Scientific notation, capital letter	6.022141E23
`%f`	Floating point	3.14159
`%o`	Unsigned octal	610
`%s`	String of characters	Sample
`%u`	Unsigned decimal integer	7235
`%x`	Unsigned hexadecimal integer	7fa
`%X`	Unsigned hexadecimal integer (capital letters)	7FA
`%4.10f`	Specific precision floating point	0014.6589743241

printf Function in C

- Note: Printing out using the wrong specifier for the type of variable can produce some very strange results

```
int32_t res = 15 ;

printf("res = %f\n",res) ;
```

→ prints res = 0.000

```
int32_t res = 15 ;

printf("res = %d\n",res) ;
```

→ prints res = 15

Review Example Code

- Review Lecture 5 example code