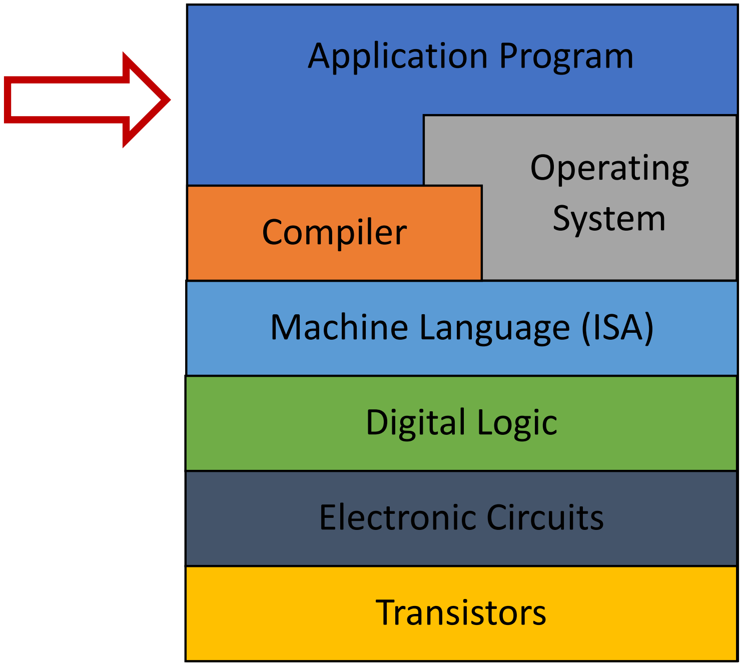
THE UNIVERSITY OF KANSAS

EECS 388: Embedded Systems

Lecture 3

Instructor: Tamzidul Hoque, Assistant Professor,
Dept. of EECS, University of Kansas
(hoque@ku.edu), office: Eaton 2038

Context


Application Program
Operating System
Compiler
Machine Language (ISA)
Digital Logic
Electronic Circuits
Transistors

Example In-class quiz question

- **True/False:**

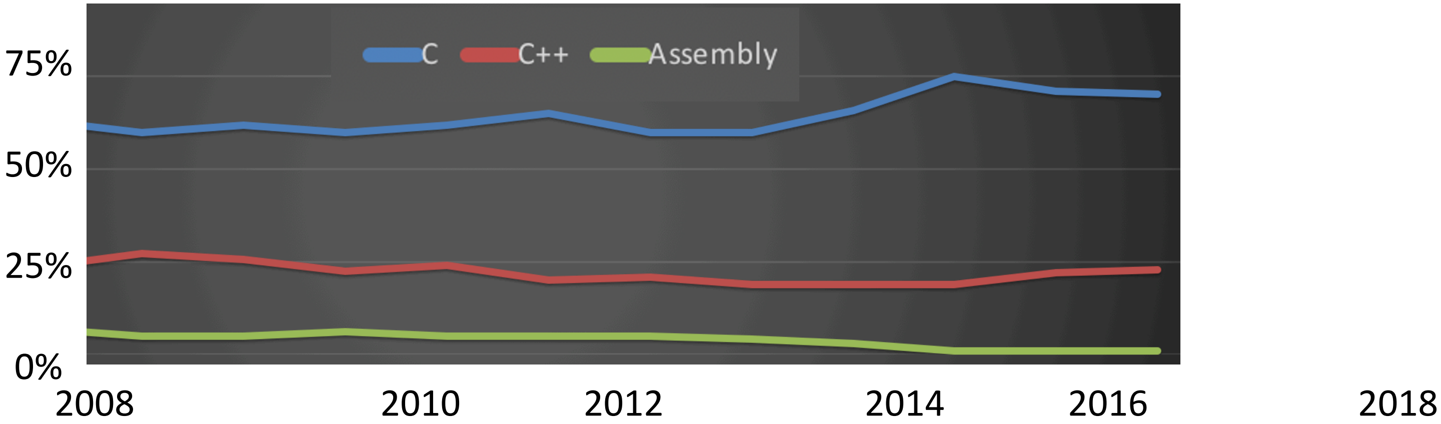
1. Assembly language programs generated from the **same** C program is **same** for different Instruction Set Architecture (ISA)
2. The same ISA can be implemented with two different microarchitectures
3. The preprocessing step of the compiler tool-chain expands the assembly code to generate a machine-readable binary
4. Embedded systems are generally not programable by end-users/ consumers

Embedded Software Market Share

C: 70%

C++: 25%

Assembly, Java, etc: 5%



https://embeddedgurus.com/barr-code/2018/02/c-the-immortal-programming-language/

Embedded systems primarily uses C programming. But WHY?

- **High-level:**
  - Feasibility of learning, portability across OS, hardware independent compared to assembly
- **Low level:**
  - Lower code overhead for execution
    - Ex: In Java, we need JVM (Java Virtual Machine) in addition to the jar files (Executable)
  - Bitwise operation (programming registers)
    - Changing register values
  - Memory management (access & allocation)
    - Pointers, dynamic memory allocation
  - I/O operation
    - Using pointers we can configure I/O devices

Bits vs Bytes

$$\overbrace{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}^{\text{8 bits == 1 byte}}$$

Most
significant bit
(MSB)

Least
significant bit
(LSB)

- To understand a value of a variable and use it properly, a compiler needs to understand both the raw values and the type of a variable

Declaring Variables

- Variable is nothing but a name given to a memory location.
- Declaration is needed before using variable in program
- Declaration need (at least) two things:
  - Data type
  - Variable name

```
<data-type> <var name>; →    int var;
```

- Variable value can be initialized during declaration

```
<data-type> <var name> = < value>; →    int var = 7;
```

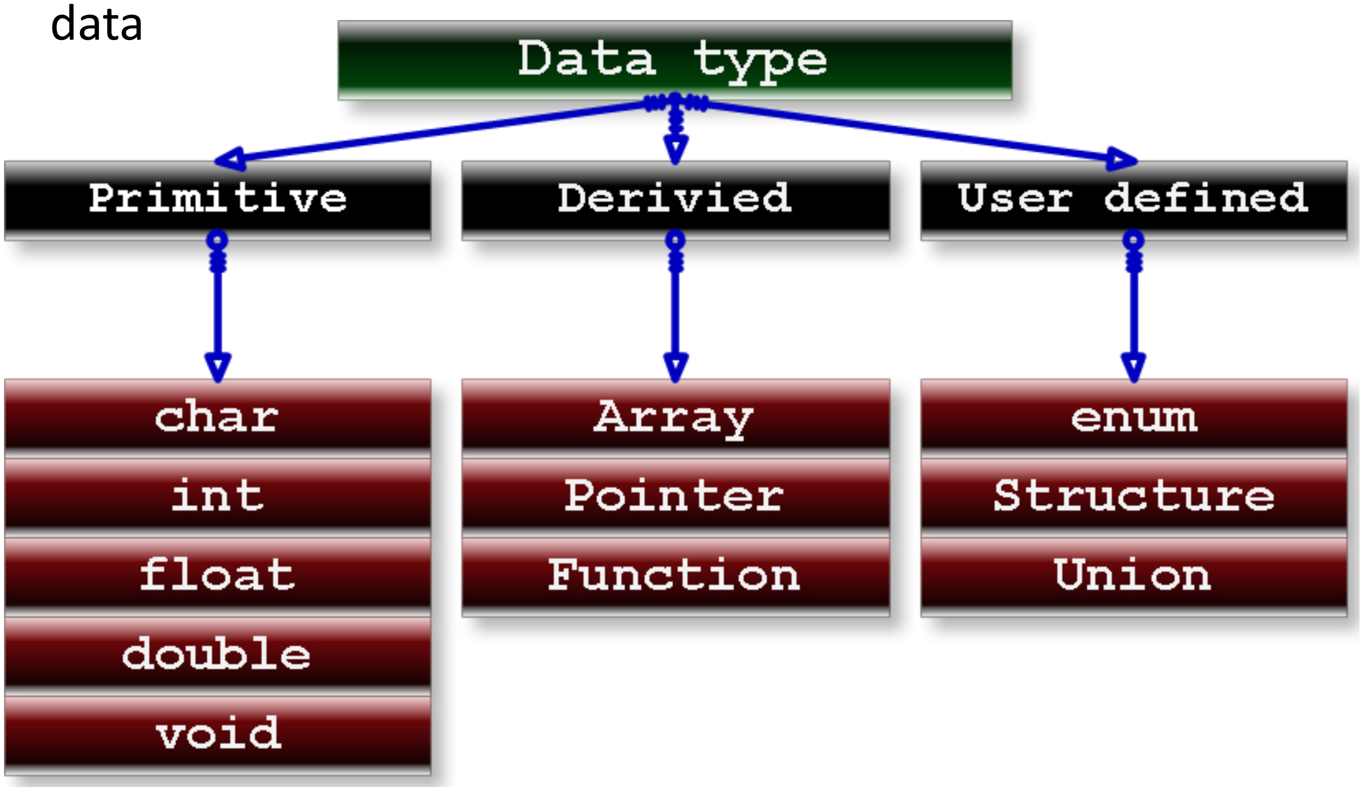
Declaring Variables

- When we declare once, we don’t need to declare the type again (within the same function):
    - int var;
    - var=13;

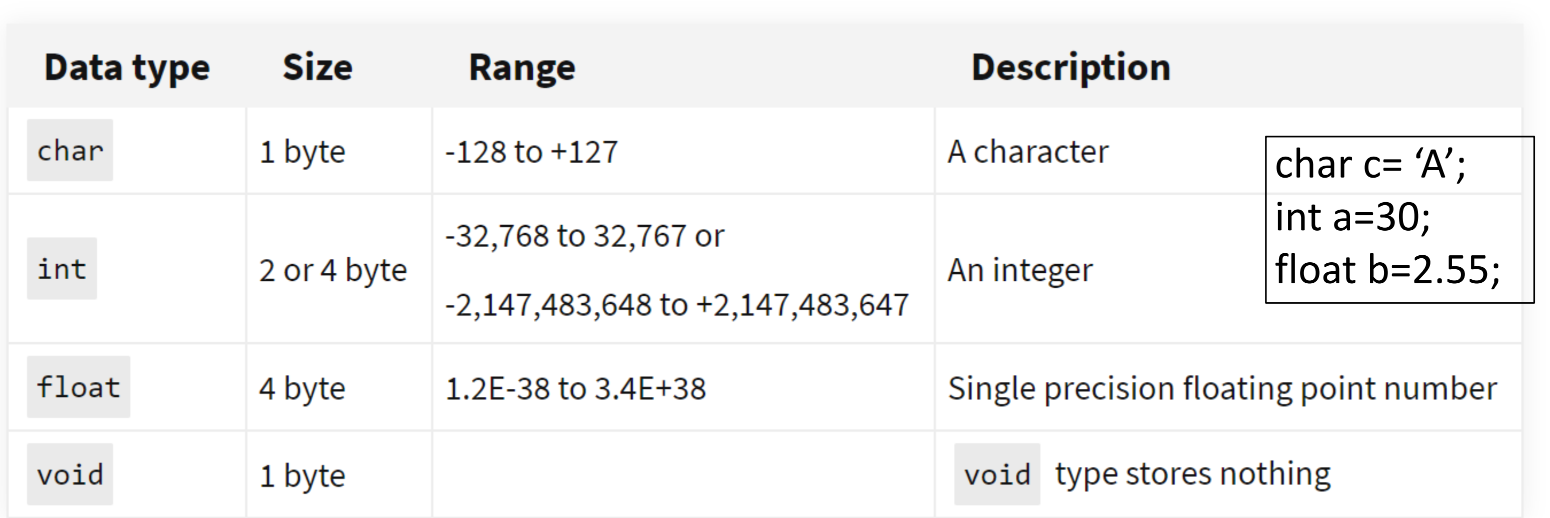
- We can assign variable to a variable:
    - Int var=13;
    - Int new_var=var;

Data Types

- Data type is a way for the programmer to tell the compiler or interpreter how the data will be used/processed
- Allows programmer to define precision, range, and sign of the data



Primitive Data Types



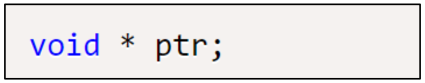
Data type	Size	Range	Description
char	1 byte	-128 to +127	A character
int	2 or 4 byte	-32,768 to 32,767 or -2,147,483,648 to +2,147,483,647	An integer
float	4 byte	1.2E-38 to 3.4E+38	Single precision floating point number
void	1 byte		void type stores nothing

```
char c= 'A';
int a=30;
float b=2.55;
```

*Size of int is compiler dependent, but mostly 4 bytes.

- **Void**
  - Function return type: when function don't return anything
  - Universal **pointer type**: placeholder when datatype is unknown

```
void func () {
            printf("Hi
EECS388!");
}
```



```
void * ptr;
```

Type Modifiers

<type modifier (s)> <data-type> <var name> = < value>;

Increase the size of data types or change their properties

- Short
- Long
- Unsigned
- Signed

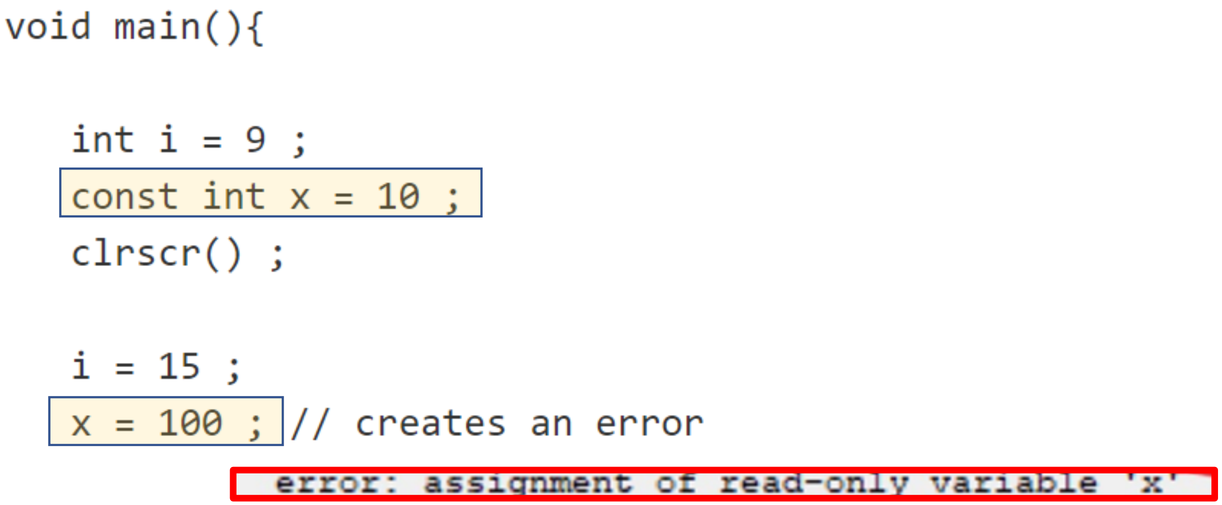
Data type	Storage	Range
char	1 Byte	[-128,+127]
unsigned char	1 Byte	[0, 255]
short int	2 Byte	[-32768,+32767]
unsigned short int	2 Byte	[0, 65535]
int	2 or 4 Byte	$[-2^{15}, 2^{15}-1]$ or $[-2^{31}, 2^{31}-1]$
long int	4 or 8 Byte	$[-2^{31}, 2^{31}-1]$ or $[-2^{63}, 2^{63}-1]$
long long int	8 Bytes	$[-2^{63}, 2^{63}-1]$

Example: unsigned long int var = 200;

Type qualifier

<**type qualifier**> <type modifier**>** <data-type> <var name> = < value>;

- The keywords which are used to modify the properties of a variable are called type qualifiers.
  - **Const:** *variable can't be changed once it is defined*
  - **Volatile**



```
void main(){

  int i = 9 ;
  const int x = 10 ;
  clrscr() ;

  i = 15 ;
  x = 100 ;// creates an error
```

```
error: assignment of read-only variable 'x'
```

Type qualifier

<**type qualifier**> **<**type modifier**>** <data-type> <var name> = < value>;

- **Volatile:** *tells the compiler that the value of the variable may change at any time--without any action being taken by the nearby code (could change by the hardware instead)*

- Example of variable with such change:
    - Registers of memory-mapped peripheral
    - Global variables modified by an interrupt service routine
    - Global variables accessed by multiple tasks within a multi-threaded application

https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword

Data Storage in Memory

$$\overbrace{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}^{\text{8 bits == 1 byte}}$$

Most
significant bit
(MSB)

Least
significant bit
(LSB)

$1000_2=8_{10}$

$0000_2=0_{10}$ Change MSB: high impact

$1001_2=9_{10}$ Change LSB: low impact

Introduction to Number Systems

represent the numbers.

- Ancient number system:
  - tally marks → not good for large numbers

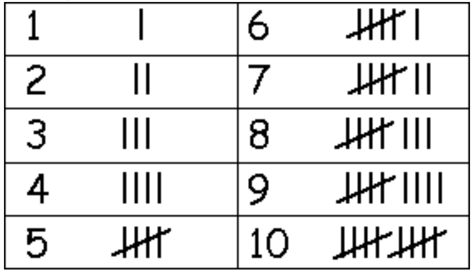


Figure: tally marks

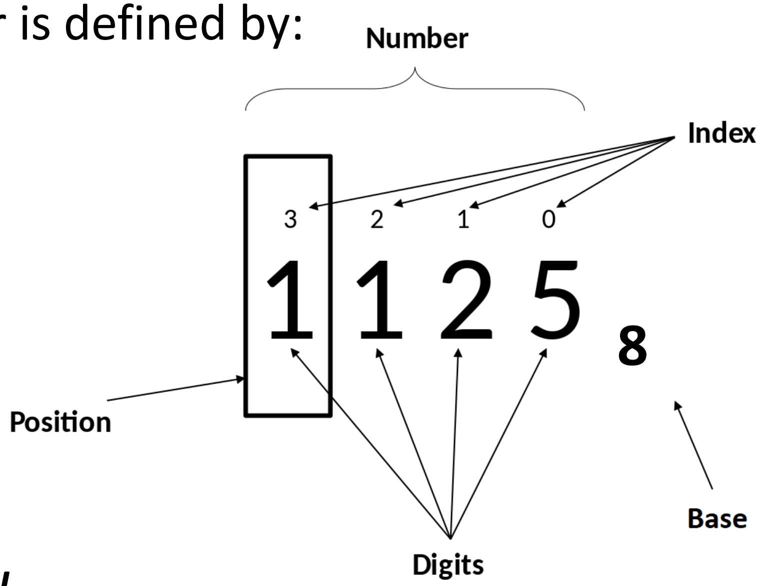
❑ **Modern numbering system: "positional system"**

- Represents numbers using an ordered set of digits
- Value of a digit depends on the "base"
- The **base** is the **number of digits** in the system

Positional System	Base	Allowed Digits
Binary	Base 2	0,1
Octal	Base 8	0,1,2,3,4,5,6,7
Decimal	Base 10	0,1,2,3,4,5,6,7,8,9
Hex	Base **?**	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (A=10, …, F=15)

Positional System

- Value of a number is defined by:
  - Digits
  - Index/position
  - Base



<u>Converting to Decimal</u>

	1	1	2	5
Digit $\times \boldsymbol{Base}^{\boldsymbol{Index}}$				
	512	64	16	5
Value= Sum of Products	Value in decimal=(512+64+16+5)=597			

Converting to Decimal (cont.)

- Binary (base 2)
  - Symbols: 0,1

$$Digit \times Base^{Index}$$

  - E.g., 1011_2 = **0b1011** = $1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

- Hexadecimal (base 16)
  - Symbols: 0,1,…,9,A,B,…,F
  - E.g., 123_{16} = **0x123** = $1x16^2 + 2x16^1 + 3x16^0$

*Assuming unsigned number

Number Systems

- Conversion to Decimal: Hand calculation is hard for large number
- Need sum($Digit \times Base^{Index}$) equation

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	🟥	🟨
9	🟥	🟨
🟥	0xA	🟨
🟥	0xF	🟨
🟥	0x1F	🟨
🟩	🟩	0b1000 0000
🟩	🟩	0b1000 0011
🟩	🟩	0b1000 0000 0000 0000

Complete Red→ Green →Yellow

Number Systems

- Conversion to Decimal: Hand calculation is hard for large number
- Need sum($Digit \times Base^{Index}$) equation

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	0x2	🟨
9	0x9	🟨
10	0xA	🟨
15	0xF	🟨
31	0x1F	🟨
🟩	🟩	0b1000 0000
🟩	🟩	0b1000 0011
🟩	🟩	0b1000 0000 0000 0000

$1F_{16}=1*16^1 + F*16^0= 16+15=31_{10}$ Complete ~~Red~~→ **Green** →Yellow

Number Systems

- Conversion to Decimal: Hand calculation is hard for large number
- Need sum($Digit \times Base^{Index}$) equation

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	0x2	
9	0x9	
10	0xA	
15	0xF	
31	0x1F	
	0x80	0b1000 0000
	0x83	0b1000 0011
	0x8000	0b1000 0000 0000 0000

Starting from LSB, each 4-bit binary is a Hex digit.

Number Systems

- Conversion between Hex & Binary: Easy even for large numbers
- Just need to know the conversion between hex to bin for 0 to F

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	0x2	
9	0x9	
10	0xA	
15	0xF	
31	0x1F	
128	0x80	0b1000 0000
131	0x83	0b1000 0011
32768	0x8000	0b1000 0000 0000 0000

Number Systems

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	0x2	0b10
9	0x9	0b1001
10	0xA	0b1010
15	0xF	0b1111
31	0x1F	0b1 1111
128	0x80	0b1000 0000
131	0x83	0b1000 0011
32768	0x8000	0b1000 0000 0000 0000

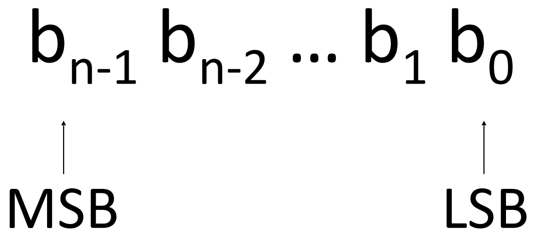
Starting from LSD, each Hex digit is a 4-bit binary

Categories of Numbers

- Unsigned
- Signed
- Fractional

Unsigned Numbers

- No sign, only magnitude/value
- n-bit binary number:



$b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0$

MSB

LSB

- Decimal value:

$$b_{n-1} * 2^{n-1} + b_{n-2} * 2^{n-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$$

- Example:

  0b10011 = $2^4 + 2^1 + 2^0$ = 19 (d)

2’s Complement Numbers

- Most common method of representing signed integers in computers
- Getting two's complement of an integer:
  - Write the number 28 in binary, e.g., 00011100
  - **Invert** the digits 11100011
  - **Add 1** to the result 11100100
  - That is how one would write -28 in 8 bit binary.
  - A leading 1 means a negative number, a leading 0 means positive in 2’s complement.

- Verify: Subtract 7 from 9 using 2’s complement
  - 7(dec)=0b0111; thus: -7= 0b1000+0b1=0b1001
  - Verify: 9 – 7 = 9 + (-7) = 0b1001 + 0b1001 = 0b0010= 2

Converting from 2's Complement to signed decimal

- convert as binary number to decimal but with a negative sign with left most binary digit.

- Decimal value:

$$- (b_{n-1} * 2^{n-1}) + b_{n-2} * 2^{n-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$$

- Example:

  $0b10011 = - (2^4) + 2^1 + 2^0 = -13$ (d)

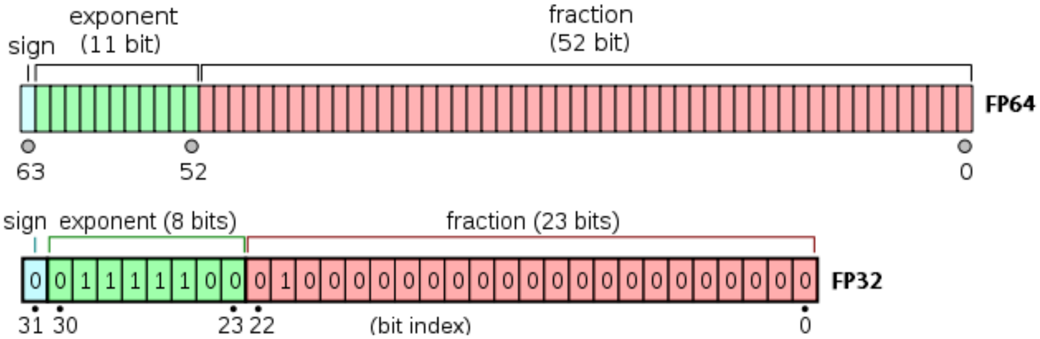
Fractional Numbers: Float and double

Float

- IEEE 754 single precision floating point numbers
- 1-bit sign, 8-bits exponent, 23-bits fraction
- 6 significant decimal digits of precision

Double

- 1-bit sign, 11-bits exponent, 52-bits fraction
- 15-17 significant decimal digits of precision



sign
exponent (11 bit)
fraction (52 bit)
FP64
63
52
0
sign
exponent (8 bits)
fraction (23 bits)
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
FP32
31
30
23
22
(bit index)
0