

# CS-118-02 Week 4 Lecture 1

Varun Narravula

2021-04-14

## Floating-Point Numbers

Floating point numbers work very similarly to scientific notation, except they are in binary and have a few more tricks up their sleeve. They also do not use two's complement, because it can be pretty inflexible. Here's a great video that can help you understand what the hell this IEEE-754 standard I'm talking about even is.

Raw link: <https://www.youtube.com/watch?v=RuKkePyo9zk>

Parts of an IEEE-754 standard 32-bit float:

- Sign :: 1 bit
- Biased exponent :: 8 bits
- Mantissa :: 23 bits

A 64-bit double type works in the same way, except it allocates more space (11 bits) to the exponent for more precision (hence, double-precision), and 51 bits to the mantissa for a larger number range as compared to floats. Usually, doubles are used unless you have extreme space constraints, but for the sake of simplicity, let's stick with floats.

So for a number 20.75, its 32-bit floating-point representation would be:

Sign	Exponent	Mantissa
0	10000011	01001100000000000000000

How do you extract these different parts? Here's a code example that does this.

```
#include <iostream>
#include <bitset>

int main() {
    float f = -7.75;

    // The magic part! This could result in undefined behavior on
```

```

// some systems, so be careful when using it. I'm using uint32_t
// to ensure that it is 32 bits, but for all intents and purposes,
// it is the same as `unsigned int`
uint32_t p = *(uint32_t *)&f;
std::cout << "Bit representation of float: " << std::bitset<32>(p) << "\n";
std::cout << "Decimal uint32_t representation of float: " << p << "\n";

// Holding loop values to save unnecessary reassignments
uint32_t mask = 0;
uint32_t i = 0;

// Extracting mantissa
uint32_t mantissa = 0;
for (; i < 23; i++) {
    mask = 1 << i;
    mantissa += p & mask;
}

std::cout << "Mantissa (in binary): " << std::bitset<32>(mantissa) << "\n";
std::cout << "Mantissa (in decimal): " << mantissa << "\n";

// Extracting exponent
uint32_t exponent = 0;
for (; i < 30; i++) {
    mask = 1 << i;
    exponent += p & mask;
}

std::cout << "Exponent (in binary): " << std::bitset<32>(mask) << "\n";
std::cout << "Exponent (in decimal): " << mask << "\n";

// Extracting sign
uint32_t sign = p & (1 << 31);

std::cout << "Sign: " << (sign == 0 ? "Positive" : "Negative") << "\n";
return 0;
}

```

The output of the above program:

```

Bit representation of float: 11000000111110000000000000000000
Decimal uint32_t representation of float: 3237478400
Mantissa (in binary): 00000000011110000000000000000000
Mantissa (in decimal): 7864320
Exponent (in binary): 00100000000000000000000000000000
Exponent (in decimal): 536870912
Sign: Negative

```

When you think about it, this can only make sense because of the conversion to an unsigned integer! All data types can be represented in terms of unsigned integers, so to get the bit representation of a type, you can just cast it (make sure you do it in a way that does not cause undefined behavior though). Then, you would loop over each bit and retrieve its value using a bitwise **and**. You loop from positions 0-23 to get the mantissa, positions 24-31 to get the biased exponent, and you just need one value for the sign, so you can directly **and** it to find the value. And there you go! You could also index the bitset, but that's also relying on an external library, so that's a no-go here. I'm only using `std::bitset` to print binary representations of numbers in C++ using `cout` instead of `printf`.

Important tangent: an 8-bit pointer (or address) can address 256 (or  $2^8$  different locations, from 0-255. For each bit, you can increase the exponent (so  $2^9 = 512$  for a 9-bit pointer). Most pointers have the same size regardless of the data it actually points to.

## The Stack

Running processes have three things associated with them:

- Stack :: memory automatically allocated for a process
- Data (or heap) :: dynamically allocated memory
- Executable code :: the actual instructions to be ran

There's a reason a stack is called a stack. The reason for it is because the data in the stack is quite literally stacked! Say you have some data in this code:

```
int main() {
    int a = 1;
    int b = 2;

    fx();
}

void fx() {
    int c = 10;
    int d = 11;
    gx()
}

void gx() {
    int e = 111;
    int f = 102;
}
```

`a` and `b` are allocated in registers. But what happens when you have to go to `fx`? What is the state of them going to be during the function? Well, what happens

is the program has to remember them in the stack! So it takes a snapshot of **a** and **b**, and pushes them onto the stack, and then it adds another section on the stack on top of this snapshot for **c** and **d** and whatever data is there! It then does its thing in the registers for **c** and **d**, and then moves on to **gx**. **c** and **d** are now on the stack, above **a** and **b**. And when **gx** and then **fx** finishes, then **c** and **d** are popped off the stack. **fx** finishes and then **a** and **b** are popped off at the end. It follows the same principle that the queue data type does: *last in, first out*.

A process that you are familiar with that works in this way is recursion. The same function gets called multiple times, and each call is stacked onto each other until the recursion ends, when the last function returns a result, then each function returns their results, and the function stack then unwinds.

There is a 64-bit register called **rsp**. What is this? It's a special register that tells you what is on the top of this stack! That way the registers know what is on the top of the stack at any given time so that whenever something is popped, it can go back. Make sure to be careful if you attempt to assign anything here other than that, it's special.

Another register that is similar is **rbp**, which is called a base pointer. This points to the very beginning of the stack, so that one can traverse the stack easily without having to do arithmetic with magic numbers and sizes of pointers and other things.

Another very important, but different register is **rip**. This points to a specific point in the **.text** section in assembly. It is an instruction pointer. It's pretty self-explanatory: it points to the instruction getting executed. If you change this, you can essentially do whatever the hell you want! This allows for potential viruses to infect even hardware code. So be careful!

Say your stack size is 1000. What is going to be there when you push something to it, like a **dword**? Well, it actually has to subtract from this stack based on the diagram. So the next memory location (for the stack) would be  $1000 - 8 = 992$ . It would go on and so on. A stack is going to be aligned to the number of bits that the architecture uses. Even though a word is 16 bits, it's going to be allocated on the stack within a 64-bit section on a 64-bit architecture.

Fun little tidbit: a stack has limited space. So what happens when a stack runs out of space? This is actually an almost-always unrecoverable error that causes a program to crash, called a **stack overflow**. I wonder where I've heard that term before?

Phew, that was a lot! Let's move on.

## Layers of an OS

There's two parts to an OS:

- Userspace mode :: all applications that a user can control

- Kernel mode :: the part of the OS that interfaces with the hardware for the userspace

How do userspace mode applications (like Chrome and even command-line utilities like `ls`) tell the kernel to do things? The kernel exposes interfaces that are called “system services” (i.e. drivers) that userspace applications can make “system calls” (or syscalls) to! Say if you wanted to write to `STDOUT`, which is a special file. There is a `write` syscall for that. `write` would make a call to a few other services (graphics, etc.) and then the kernel would execute the hardware details for you! It’s a layer of abstraction. Other system calls exist, like `fopen` for files, `fallocate` for allocating disk space (not memory), and others, and the list grows as the kernel grows.

Important terminology: a thread is simply the flow of execution of a program. Multithreading like you know on your CPU (CPUs with multiple cores) just mean that multiple flows of execution are happening at the same time! For example, HTTP calls (to google.com or something like it, GET, POST, whatever) have to do multiple things. That’s where multithreading comes in.

You’ve already used syscalls in assembly code! Let’s go over the exit routine for a program. In assembly, it’s very common:

```
exit:
    mov rax, 60; magic number for exit system call
    mov rdi, 0 ; exit status 0
    syscall ; performing the syscall
```

The comments give it away. `rax` is a register that the `syscall` instruction reads and references in the Linux kernel. There’s a series of hardcoded numbers that correspond to different syscalls (here, 60 corresponds to `exit`) in the kernel. When you move 60 into `rax`, and run `syscall`, the kernel runs `exit`, and it looks at `rdi`, cleans up the program, and emits the exit status from `rdi`.

## Carrying and the Flag Register

Another cool register is called the flag register (often referred to as `rflags` or `eflags`). This register is special, and actually cannot be written to in the normal way with `mov` or `add`. It can only be accessed by special instructions because it holds important flags. Say if you wanted to add two digits, but the result is three digits. What would you need to do? You have to carry the digit and make a new one. If you didn’t, something like  $60 + 60$  would result in 20, rather than 120, because the third digit doesn’t fit in 2.

Let’s have some code:

```
mov al, 255
add al, 1
```

This would actually be 0! Why? Look at the bit pattern: 1111 1111 is 255, right? Well, once it overflows past, the bit pattern of 256 would be 1 0000 0000,

but that first 1 would not fit into an 8-bit register. So it's stripped out. This is called an overflow. Yes, it's called an overflow. There's lots of different types of overflows, do keep track of them.

When a register overflows, a carry bit (CF) is set in **eflags**. Other flags are also set in it, such as the zero flag (ZF), the overflow flag, and others. The **pushf** instruction pushes all of the flag register on the stack, and has no operators. Since the flag register is on the stack, now you can pop it. To actually check if the bit is set, each bit is at a hardcoded position, so use **and**! It's an instruction, same as C's and C++'s **&**. **and** takes two operands, and it puts the result in the first operand. To detect if it's 1 at bit position zero, you can just **and** the result from **pop** with 1.

There's a few different versions of **pushf**, just with different sizes. **pushfd** does a 32-bit carry value, and **pushfq** does a 64-bit value. It's in the names, similar to **qword** and **dword**. The same variations exist for **popf**.

There is something you need to remember called "parity", which makes sure that the bit is actually what it is supposed to be. It's like a checksum for a file, if you know what that is. Checksums are a type of parity. You check the checksum to see if it is accurate. The parity bit is set differently depending on the implementation, but on x86\_64 processors it's set on every arithmetic operation.

Extra note: **clic** clears the carry flag, while **stc** sets it. You shouldn't really need to set the carry flag manually right now at least.

That's pretty much all I got for today, I did lose track of what he was saying sometimes, especially near the end, so if I missed anything just let me know.