CS-118-02 Week 4

Varun Narravula

2021-04-14

Floating-Point Numbers

Floating point numbers work similarly to scientific notation, except they are in binary and have 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:

1. Sign :: 1 bit
2. Biased exponent :: 8 bits
3. 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. You should use doubles unless you have extreme space constraints, but for the sake of simplicity, let's stick with floats.

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 <bitset>
#include <iostream>

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 32 bits, but it's 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";

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

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

    // Extracting exponent
    uint32_t exponent = 0;
    for (int i = 23; i < 31; i++) {
        exponent += p & (1 << i);
    }

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

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

    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 makes sense because of the conversion to an unsigned integer. You can represent all data types unsigned integers, so to get the bit representation of a type, you can 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 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 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 to 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 sections associated with them:

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

There's a reason a stack has its name. 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;
}
```

The CPU allocates `a` and `b` in registers at first. 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 the CPU pops `c` and `d` off the stack. `fx` finishes and then it pops `a` and `b` 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 more than once, and each call gets 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's 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 the CPU pops something, 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`; this is a base pointer. This points to the beginning of the stack, so that one can traverse the stack without having to do arithmetic with magic numbers and sizes of pointers and other .

Another important, but different register is `rip`. This points to a specific point in the `.text` section in assembly. It's an instruction pointer. It's pretty self-explanatory: it points ot 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. 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 and so on. A stack aligns to the number of bits that the architecture uses. Even though a word is 16 bits, it's 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?

Layers of an OS

There's two parts to an OS:

1. Userspace mode :: all applications that a user can control
2. 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 operate? The kernel exposes interfaces 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. The `write` syscall exists for that. `write` would make a call to some 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 the flow of execution of a program. Multithreading like you know on your CPU (multicore CPUs) mean that more than 1 flow 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 more than one thing. That's where multithreading comes in.

You've already used syscalls in asssembly code. Let's go over the exit routine for a program. In assembly, it's 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 the flag register (often referred to as `rflags` or `eflags`). This register is special, and you actually cannot write to it in the normal way with `mov` or `add`. The one way to read/write to `eflags` is by using 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 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, the CPU sets a carry bit (CF) in `eflags`. It can also set other flags, such as the zero flag (ZF), the overflow flag, and others. The `pushf` instruction pushes the entire 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, 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 `and` the result from `pop` with 1.

There's different versions of `pushf` with different sizes. `pushfd` does a 32-bit carry value, and `pushfq` does a 64-bit value. It's in the names, like `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 correct. It's like a checksum for a file, if you know what that is. Checksums are a variant of parity. You check the checksum to see if it's accurate. The parity bit can have a platform-specific implementation, but on x86_64 processors it's set on every arithmetic operation.

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

Reading Flags

How do we read flags from the flag register? You guessed it: using an `and`. Here's a code example:

```
mov al, 0ffh
mov al, 1 ; causes overflow
pushfq
pop rdx
and rdx, 1 ; checks if the CPU set the carry bit.
```

Since each architecture hardcodes bit positions for each flag, it's as easy as creating a mask and then `and`ing it with the eflags register. But, you stil have to pop it first using `pushf` and `pop`ping into a register. It would be clearer to read if the mask is in binary notation, but the mask is 1, so it does not make a difference.