HACKING THE METAL II

HARDWARE AND THE EVOLUTION OF C CREATURES

A Def Con 30 Workshop
by @eigentourist

TOOLS FOR THE WORKSHOP

- A laptop computer running Windows, or running a Windows VM
- The Netwide Assembler (NASM) v2.15.05, available here
- Visual Studio 2022 (Free community edition available here)
- Optional: Visual Studio Code - available here
- Optional: IDA Disassembler Free Edition – available here

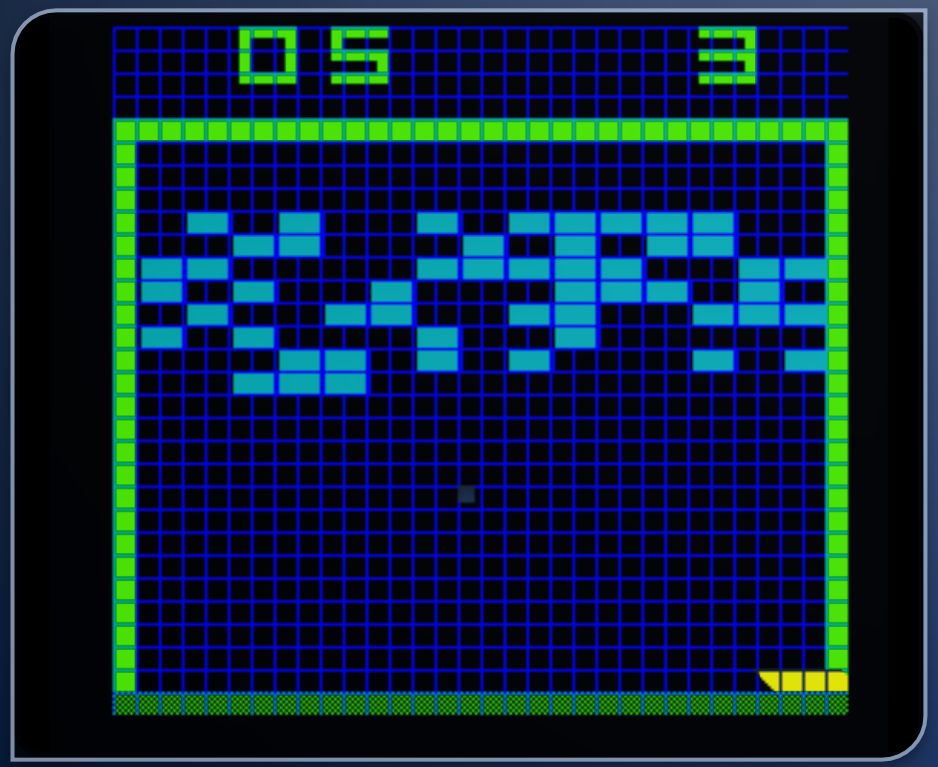
WHY THE NETWIDE ASSEMBLER AND NOT MASM?

- The syntax for assembly language in NASM is cleaner and more readable (in this humble programmer's opinion)
- NASM is available on other platforms in addition to Windows, allowing you to write assembly code in Linux or Mac OS without having to climb another learning curve
- When you install Desktop Development With C++ tools as part of your Visual Studio 2022 installation, MASM will be included (refer to this tutorial for details) – so if you find that you prefer MASM, you'll have ample opportunity to migrate while continuing to use the tool set from this workshop

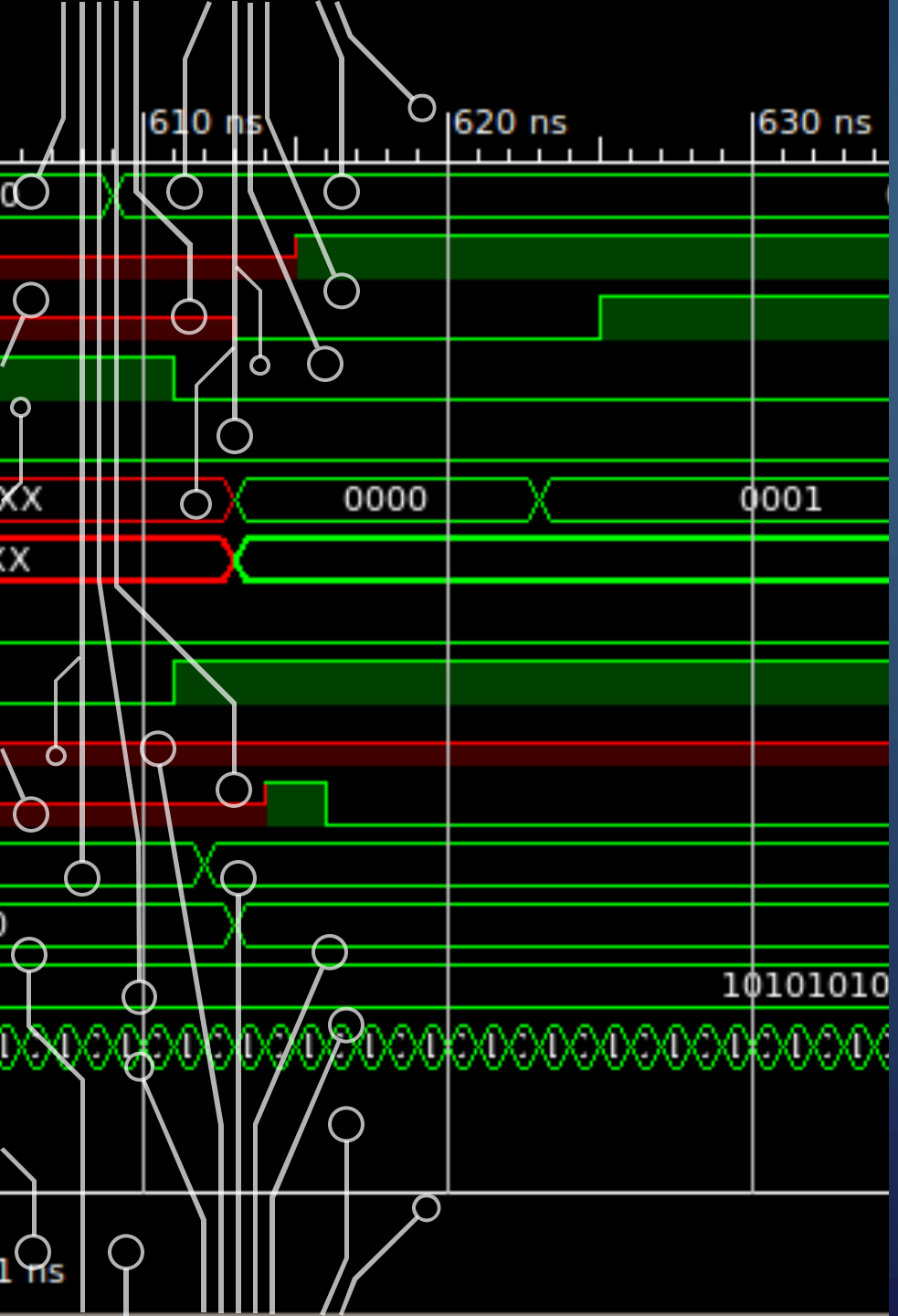
RETURNING TO THE METAL

- The first Hacking The Metal workshop at Def Con 29 introduced the Def Con audience to assembly language programming
- This year, as we celebrate a 30-year hacker homecoming, the sequel workshop will dive down to the hardware beneath assembly language – we will design CPU components using a hardware design language called Verilog
- Moving up through assembly language again, we will then transition to programming in C – not treating it as a high-level language but understanding it as a portable shorthand for assembly.

HARDWARE – COMPUTING ON THE METAL



- Where are the tools for designing computing hardware? They weren't listed in the second slide!
- The answer is here: a free web-based IDE that offers a Verilog workbench as part of a suite of tools for building and running retro games that were built directly with digital logic (think Pong and Breakout) or with assembly language for vintage consoles like the Atari 2600, all built by Steven Hugg (Twitter: @sehugg)

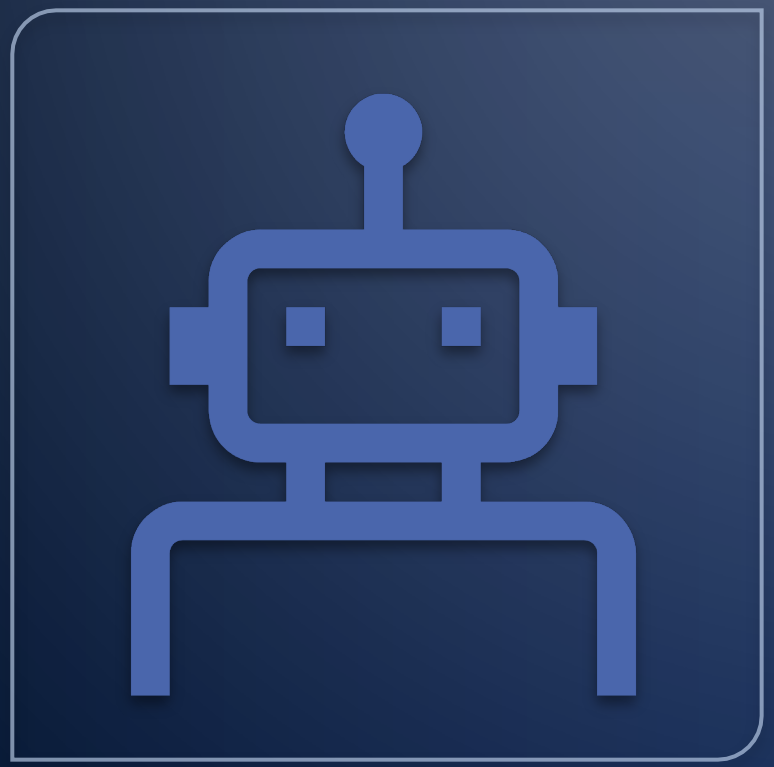


HARDWARE – COMPUTING ON THE METAL

- Using this IDE, we will load and test code written in a hardware description language called Verilog, which allows us to describe logic circuits that will store and manipulate bits, creating the basis for a CPU.

- Eventually, we will tour and examine Verilog code that implements an Arithmetic Logic Unit (ALU), and then incorporates it into a design for a working CPU, which operates according to the classic Von Neumann architecture, as all CPUs do, to this very day.

WHY HARDWARE DESIGN?



Don't panic! This isn't a call for everyone to stop doing your thing and become an electrical engineer.

Ultimately, there's a secret that all programmers discover sooner or later as they write code in their favorite language: the "machine" you are programming for… doesn't really exist.

It's a simulation – created and perpetuated by something beneath, and if we want to understand how to get the best results out of our favorite tools, we do well to understand the simulation… and that thing underneath that keeps it running.

LIVING IN THE SIMULATION

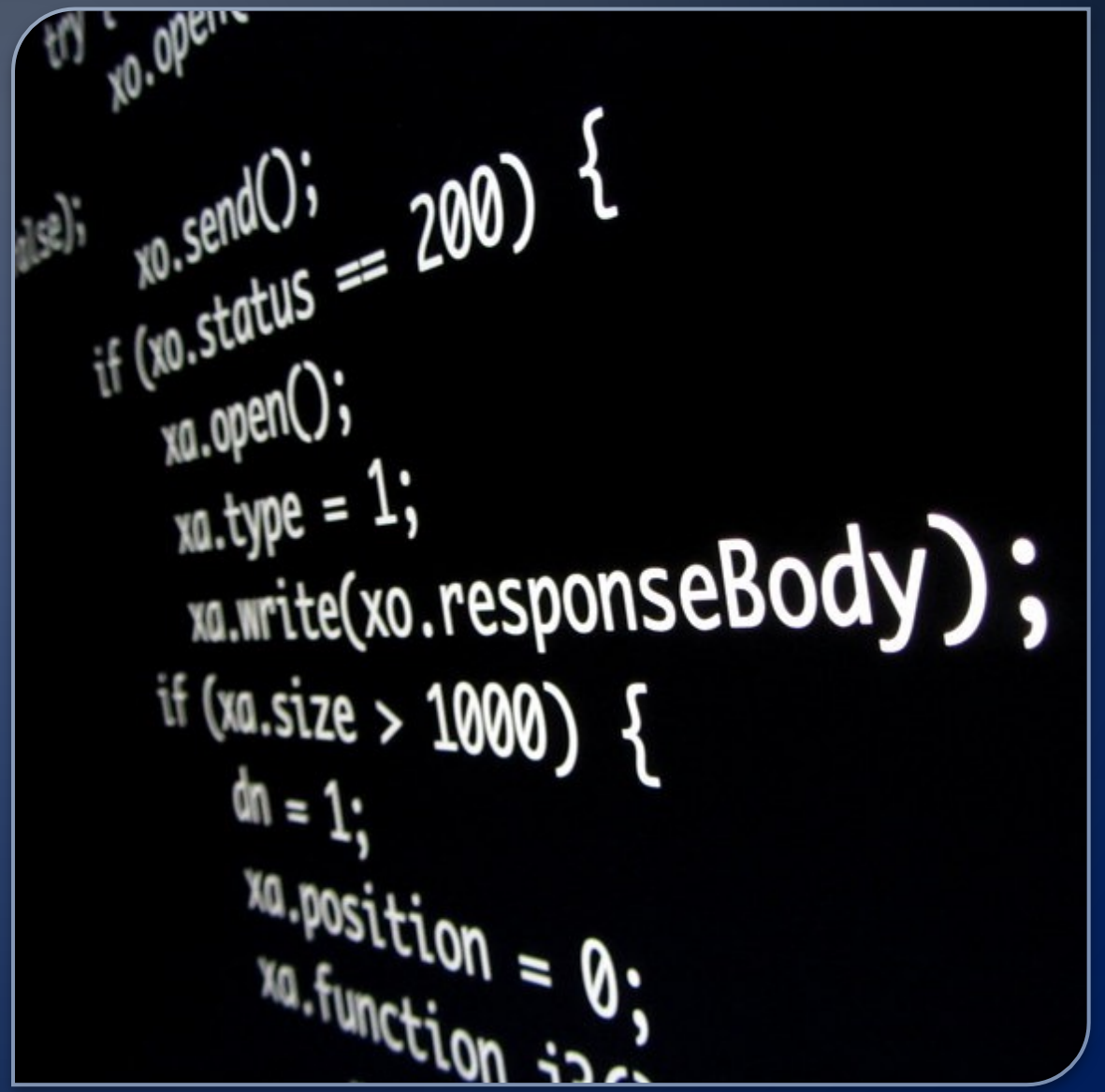
Let's say you're doing JavaScript – either for the browser or for the back end with something like Node.js – what beast is executing your code?

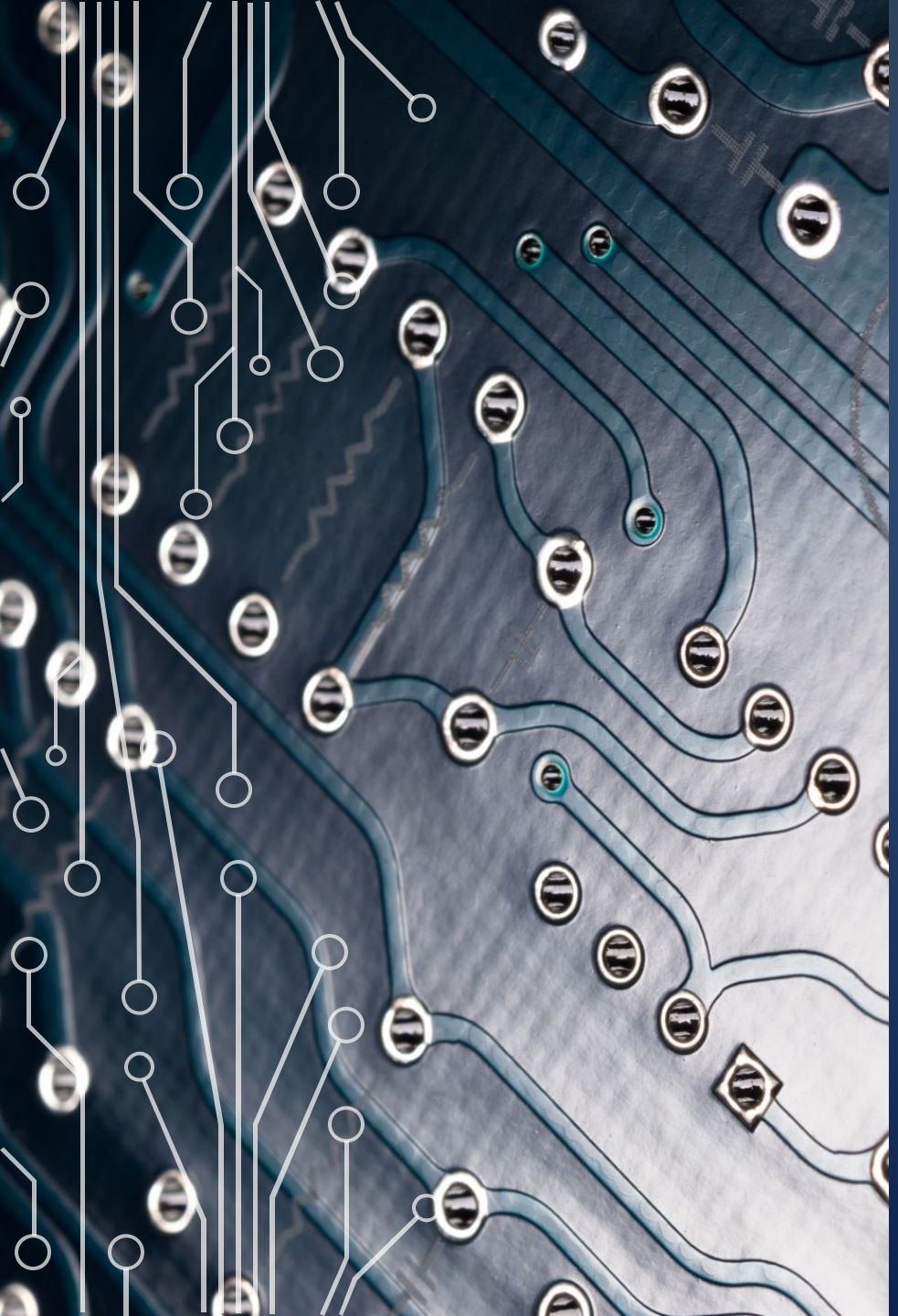
It's a JavaScript engine, and isn't that the end of the conversation?

Not really – that engine that runs your code is itself executing on… something.

Odds are fair that it's executing in a container, or on a VM, or on a physical machine, in which case it's being executed by a machine that understands how to execute machine code.

But *what* is that machine code executing on?





LIVING IN THE SIMULATION

The truth is, every time you write something like

```
mov rax, 0
```

…in assembly language, you can do that because – at the very least – someone built a set of digital logic circuits that will do the task of moving the value *zero* into the *rax* register.

In modern processors, there are often layers of microcode in between the machine code that the assembler generates from the above instruction and the actual digital circuitry. But the principle is the same: it's all simulation.

SO, IS IT TURTLES ALL THE WAY DOWN?

Not quite. But the reality is, when you write

```
mov rax, 0
```

…the eventual result is that the voltage in some tiny circuit drops down into a range that will be reliably measured as low by other mechanisms, and that allows other things to happen. And ultimately, in our favorite editor, we can write

```
int a = 0;
```

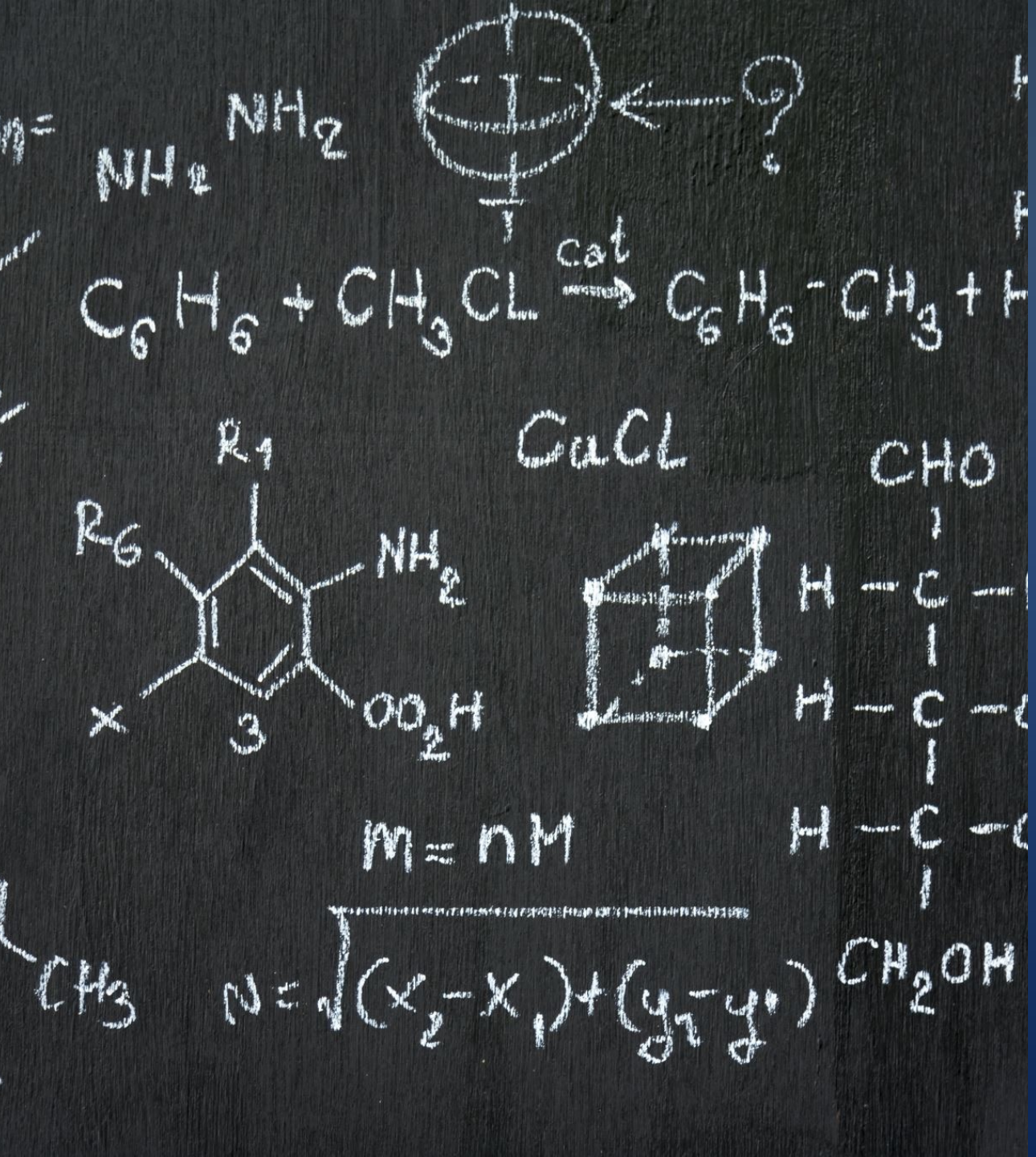
…and a whole bunch of other code, some of which may test the value of a and do this or that based on the result, and compile it…

SO, IS IT TURTLES ALL THE WAY DOWN?

…and when we run the executable binary, two amazing things happen:

Inside the machine, voltage rises and falls inside billions of electronic circuits.

But on our screens, those voltage oscillations trigger pixels to light up in coordinated patterns, emitting light in the form of an image, which means something to us: a stream of text, a set of images… something that has meaning to the human mind.

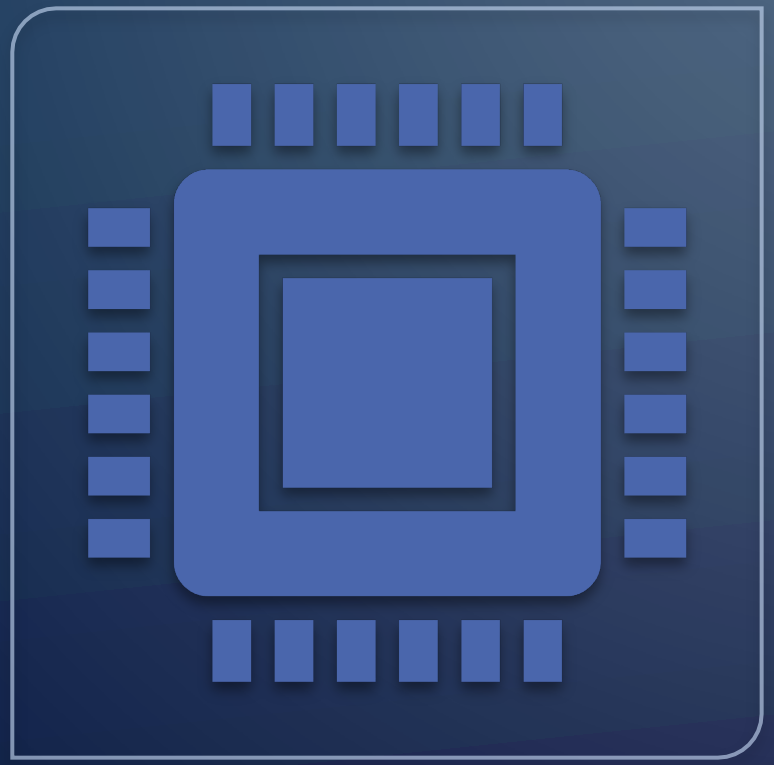


SO, IS IT TURTLES ALL THE WAY DOWN?

And amazingly, that meaning to us is the same as it would have been if hundreds of humans had scribbled on notepads and chalkboards for thousands of hours and compiled all their scribbles together and written down their observations and drawn pictures to illustrate their conclusions.

But in the machine, it all happens much, *much* faster.

THE MORE TURTLES, THE BETTER!



So, once we have a feel for what the CPU that we're coding for is doing and how it really operates… and we start to appreciate that it's far easier to write assembly language than translate instructions into their machine code bytes by hand… it's only natural to wonder if there's a way to write code that gives us a nearly equal level of control over the environment but allows us to abbreviate things a bit. With all these turtles, after all, what's the harm in having one more?

THE MORE TURTLES, THE BETTER!

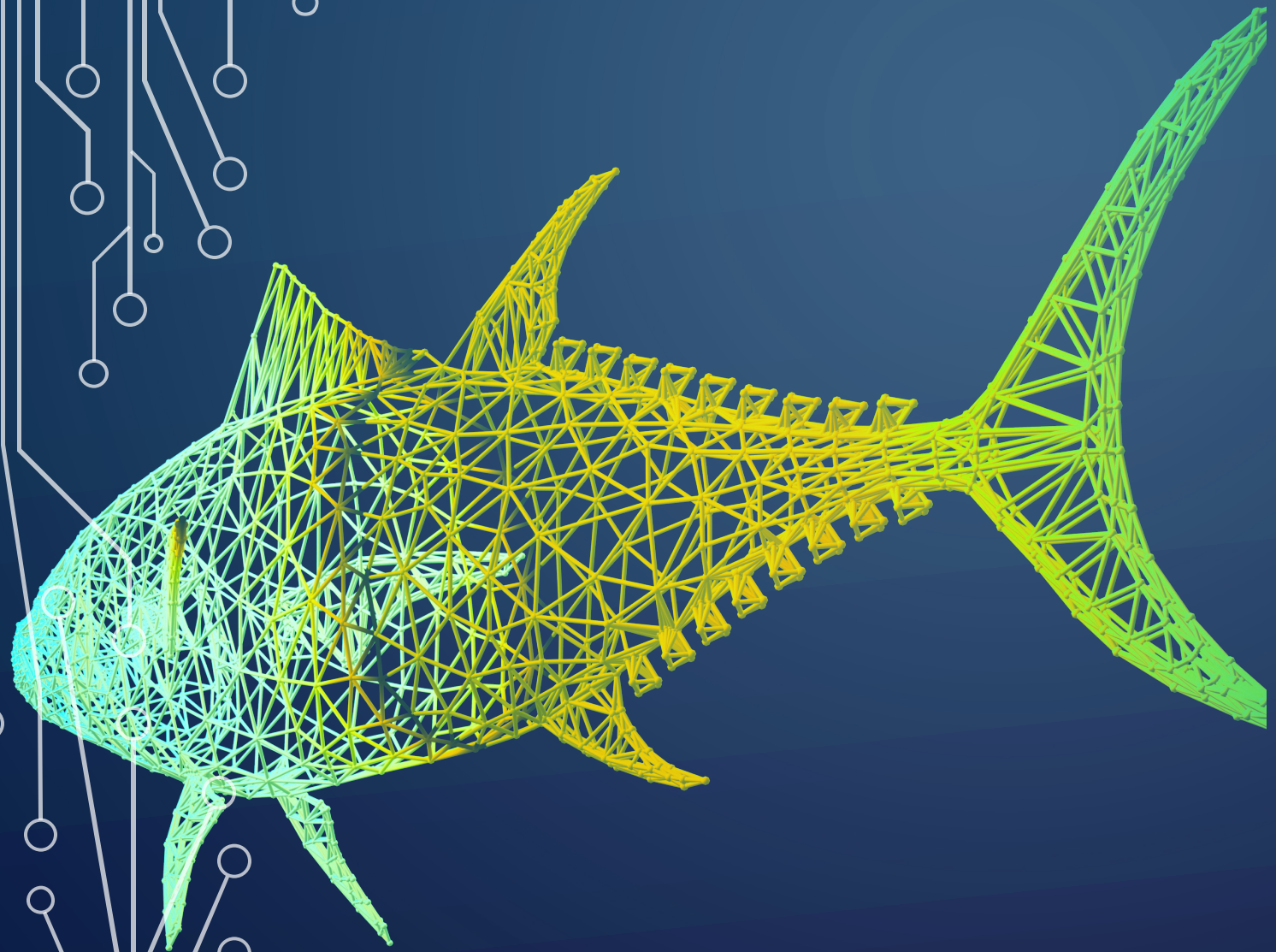
As it happens, the C programming language is a pretty good fit for being a portable sort of assembly language.

There are places where you might hear C referred to as "high level", but in this workshop we will make the case that it is not (and should not) be viewed that way.

THE MORE TURTLES, THE BETTER!

We are going to pick up where we leave off in assembly language and move forward to write C code that does the same thing.

Along the way, we will coax the C compiler to generate assembly code as it compiles. By doing this, we're going to see how often C commands translate into surprisingly few machine instructions. We will also be able to compare the assembly code generated by the compiler to hand-written assembly code that does the same thing.



THANK YOU

…for your interest in this workshop! And if you have any questions, feel free to speak up on the Discord server, which you can reach by following this invite link: https://discord.gg/xFEUFYP533

See you there!

- @eigentourist