Assembly

BY [CLARISSA LITTLER](https://www.kidscodecs.com/author/clarissalittleror/) / [LANGUAGE OF THE MONTH](https://www.kidscodecs.com/sections/all-stories/language-of-the-month/) / [APRIL 2019 ISSUE](https://www.kidscodecs.com/issues/2019-april)

It’s time for a throw back to old school programming. Dive into the nuts and bolts of coding instructions!

Last issue we talked about C, a language without a runtime that—despite being 40 years old—is still used all the time in programming operating systems, microcontrollers like the Arduino, and anything where you need a language that’s “close to bare metal” as some programmers say.

Well, in this issue we’ll be talking about the language, or rather languages, that are as close to bare metal as possible: assembly languages!

Assembly languages are the last human readable programming languages before the actual bit-strings that are the instruction codes to the processor itself. Every different family of processor designs has its own assembly language.

There are two major families of assembly languages/processor designs today: x64 (sometimes also called x86-64 or specifically Intel 64 or AMD64 if you want to name the manufacturer) and Arm. x64 includes the AMD and Intel chips you find in your desktop, laptop, or even most gaming consoles—with the notable exception of the Switch!

The other major family is Arm. Arm chips tend to be less powerful but use less power and generate less heat. These are the kind of processors you find in cell phones, tablets, and—as you might have guessed—the Nintendo Switch!

All assembly languages have things in common because all modern processors have things in common. They all have operations for doing arithmetic, they all have some way of jumping to different places in the code, they all have some way of making choices about what code to run next, and they all have instructions for interacting with memory. They also all have ways of interacting with registers.

Registers are an important concept when we’re talking about how processors work, so let’s spend a little time on the concept.

Registers are the fastest form of storage in the entire computer! They’re basically storage built right into the CPU itself that can be accessed during a single clock cycle. A clock cycle is what’s being measured when you see processor speeds described in GHz. 1 GHz means “one billion times per second” and when you say something like “this processor has four 2 GHz cores” you’re saying that there are four processing units that each run at two-billion clock cycles per second.

For a 2 GHz processor, performing an operation on a register takes 0.5 billionths of a second. Well, since you can both “read from” and “write to” the same register in a single clock cycle, like if you’re incrementing a number by one, it’s even faster than that. That’s incredible! On the other hand, RAM is at best around ten times slower.

Each core of a processor has its own registers, but each register can only hold a tiny amount of data like 64 bits. Since a processor likely has at most a couple of dozen registers, this means that all the memory on the register amounts to maybe 1kB.

All this to say that operations on registers are important and that’s why every instruction set/assembly language is mostly made up of very fast operations that act on registers only rather than on the RAM!

Assembly languages tend to be pretty complicated to get started with and both Arm and x64 have their quirks. So rather than spend time trying to introduce either Arm assembly or x64 assembly I’m going to link you to my own assembly language for a simplified, fake processor.

I’m calling this fake architecture/language Asmar, which I pronounce like ASMR — the genre of videos and streams of people quietly talking and making noises into the microphone to be relaxing. Tiny sounds. Tiny language. It makes sense in my head, at least!

Now a full treatment of even Asmar is beyond the scope of what we cover in this article but I’ll explain a few of the concepts that Asmar has in common with real assembly languages. To start let’s look at a program in Asmar. This calculates the factorial of 4:

|  |  |  |
| --- | --- | --- |
| 1 | Jmp main | |
| 2 | .fact |

|  |  |  |
| --- | --- | --- |
| 3 | MovI 1 r15 | |
| 4 | .loop |

|  |  |  |
| --- | --- | --- |
| 5 | JCon r0 body endfact | |
| 6 | .body |

|  |  |  |
| --- | --- | --- |
| 7 | Mul r0 r15 r15 | |
| 8 | SubI 1 r0 r0 |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | Jmp loop | | |
| 10 | |  |

|  |  |
| --- | --- |
| 11 | .endfact |
| 12 | JmpR r14 |

|  |  |
| --- | --- |
| 13 |  |
| 14 | .main | |

|  |  |  |
| --- | --- | --- |
| 15 | MovI 4 r0 | |
| 16 | Pc r14 |

|  |  |  |
| --- | --- | --- |
| 17 | AddI 3 r14 r14 | |
| 18 | Jmp fact |

|  |  |
| --- | --- |
| 19 | Print r15 |

How do you read this program? The first thing to note is that there’re two kinds of lines: ones that start with a . and ones that start with a capital letter. The ones that start with a dot are labels that mark lines of code so you can jump to them. The ones that start with a capital letter are the instructions!

* Jmp immediately jumps execution to a label
* MovI takes a number and puts it in a register. There are 16 registers in Asmar, labeled r0 through r15
* Pc records the program counter, the instruction the program was currently on, into a register
* AddI and SubI take a number and two registers. You can read them like:
  + AddI n rx ry computes rx + n and stores it in ry
  + SubI n rx ry computes rx – n and stores it in ry
* Mul multiplies the contents of two registers and stores it in the third
* Print prints the contents of a register to the screen
* JCon takes a register and two labels and jumps to the first label if the register contains a number greater than 0 and otherwise jumps to the second. This is our if-statement!
* JmpR jumps to the line number (after you ignore labels and blank lines) contained in the register it’s given. This is really useful for implementing function calls!

The idea of this program is that it uses r15 as the result of the function call to factorial, uses r14 to record where it jumps back to once the function is called, and puts the argument to the factorial function in r0.

So go ahead and head on over to my GitHub repo for the Asmar interpreter, instructions how to download/run it, and more detailed explanation of the language!