The basics of Arm64 Assembly

Just one instruction at a time!

Diego Crespo
Mar 9

This post is geared towards beginners, but don't worry if you don't understand all of it at once. It might take a couple of rereads or some extra googling. That's okay! Just don't forget to ask questions if you get stuck📌

Wooden Learning Block Lot on Desk by Digital Buggu

Programming in assembly is not as hard as it seems. While it's true that one line of code in a high level language will usually produce many assembly instructions, you'd be surprised at how much you can read once you get the hang of it. Our platform of choice will be a Raspberry Pi 4 running the latest version of Raspberry Pi OS.

Learning the basics 🎓

One of the simplest things you can do in assembly is just to move a value into a register. Create a file called mov.s using your text editor of choice, and add this to it.

```
/* This is a multi-line comment
   that spans multiple lines.
   It can be used to provide more
   detailed information about the code. */

.global _start // This is an inline comment. I can also use ; or @
instead of //

.section .text

_start:
  MOV X0, #0
```

The symbols below are called comments. They are skipped by the assembler when they are encountered. If you have ever programmed before some of them might be familiar to you.

```
/* */ ; // @
```

.global _start tells Linux where the beginning of our program is. If you've used other curly brace languages this is usually called main. .section .text is where our code will live. The MOV instruction is a mnemonic for a machine code instruction that is executed on the processor. In our example we move the immediate value #0 into the register X0. Numbers are usually prefixed with "#" but our assembler won't complain if you forget to add it. Since humans are terrible at writing in 1s and 0s these mnemonics help aid us when programming. To compile, link, and execute the program you must run these commands…

```
as -o mov.o mov.s
```

```
as -o mov.o mov.s

ld -o mov mov.o

./mov
```

If you didn't make any mistakes nothing should happen. We moved a value into a register but didn't do anything with it. That's okay for now we are just learning the basics.

The 'as' command assembles .s files into object files. Object files are just files with machine code instructions that can be understood by the CPU. The 'ld' command links the object file with any libraries, and other assembly files you may be using. At the end of these two steps, you are left with an **E**xecutable and **L**inkable **F**ormat file (Elf) which is the Linux equivalent of a .exe file. Let's look at another simple example. We will place it in a text file called add.s

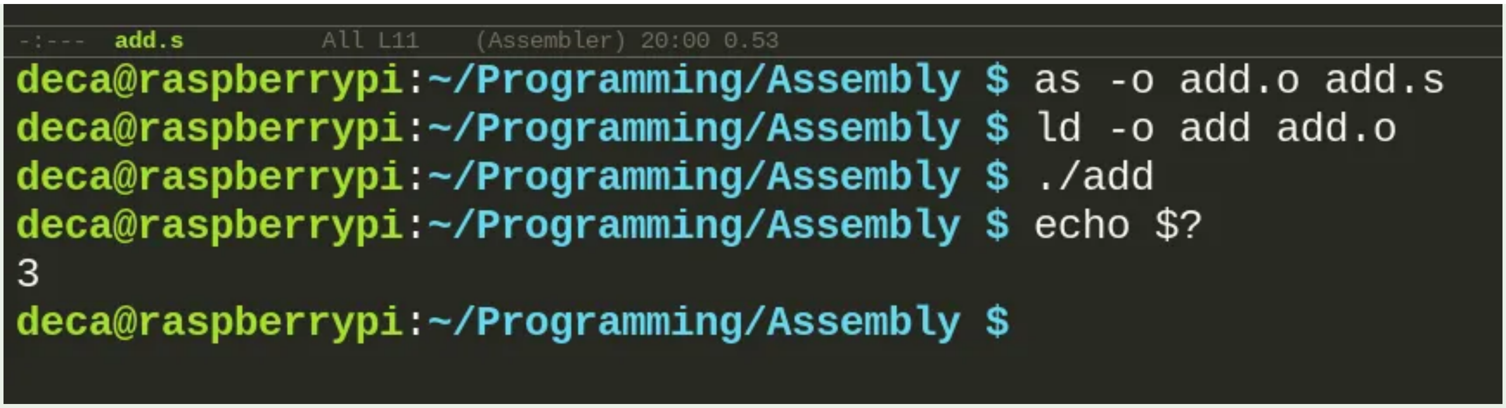
```
.global _start

.section .text

_start:
        MOV X1, #1
        MOV X2, #2
        ADD X0, X1, X2
        MOV X8, 93
        SVC 0
```

after assembling, linking, and executing this new file we will type one final command. If you type echo $? you should see the number 3 appear in the terminal.

Side note if you are using PowerShell on the Raspberry Pi like I do[1] *then you would type $LASTEXITCODE instead.*



assembling, linking, executing, and getting the exit status

In our add.s file I've introduced three new concepts. One is the add instruction, that takes numbers and registers, adds them together, and stores them in another register. The other two go together.

```
        MOV X8, 93
        SVC 0
```

What we see here is a syscall. There are registers that are treated specially by Broadcom, the makers of the Raspberry Pi 4's BCM2711 System on Chip (SOC). The 2711 SOC contains 30 general purpose registers which can be used to store values. The rest are special registers like the Stack Pointer (SP), Frame Pointer (FP), Link Register (LR), and Status Register (SR). These registers values are changed based on side effects of assembly instructions. We will see a few of these registers today.

The Linux Kernel also treats certain registers as special. Register X8 is the syscall register. When special values are loaded into this register a kernel system call can be placed by calling the **S**uper**V**isor **C**all (SVC) assembly instruction. When this happens the values in registers X0-X4 are treated as arguments to that system call depending on the syscall. In our program, we moved the value 93 into register X8 which corresponds to the exit system call.

Exit terminates our program and returns its status to the operating system. If you've ever programmed in a language that has a line like return 0, or return SUCCESS at the end of main this is what it's doing. Typically, Linux treats 0 as a successful exit, 1 and an unspecified error, 127 when an executable is invoked but not found, etc.

error, 127 when an executable is invoked but not found, etc.

As you can see, when writing assembly, we have two keep track of two major things that we don’t normally have to when programming. What the hardware is capable of, and what the operating system is capable of. In programming languages like C or Python these details are abstracted away from you to varying degrees. When writing assembly these details are necessary to write programs. The header file that documents all the syscalls for arm64 is located in the kernel source at linux/include/uapi/asm-generic/unistd.h[2] But I much prefer the format listed for ChromiumOS[3]

arm64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%x8	arg0 (%x0)	arg1 (%x1)	arg2 (%x2)	arg3 (%x3)	arg4 (%x4)	arg5 (%x5)
0	io_setup	man/ cs/	0x00	unsigned nr_reqs	aio_context_t *ctx	-	-	-	-
1	io_destroy	man/ cs/	0x01	aio_context_t ctx	-	-	-	-	-
2	io_submit	man/ cs/	0x02	aio_context_t	long	struct iocb * *	-	-	-
3	io_cancel	man/ cs/	0x03	aio_context_t ctx_id	struct iocb *iocb	struct io_event *result	-	-	-
4	io_getevents	man/ cs/	0x04	aio_context_t ctx_id	long min_nr	long nr	struct io_event *events	struct __kernel_timespec *timeout	-
5	setxattr	man/ cs/	0x05	const char *path	const char *name	const void *value	size_t size	int flags	-
6	lsetxattr	man/ cs/	0x06	const char *path	const char *name	const void *value	size_t size	int flags	-
7	fsetxattr	man/ cs/	0x07	int fd	const char *name	const void *value	size_t size	int flags	-
8	getxattr	man/ cs/	0x08	const char *path	const char *name	void *value	size_t size	-	-
9	lgetxattr	man/ cs/	0x09	const char *path	const char *name	void *value	size_t size	-	-
10	fgetxattr	man/ cs/	0x0a	int fd	const char *name	void *value	size_t size	-	-

ChromiumOS Syscall table

“But ChromiumOS is a different operating system?”, you may say. While this is true the underlying kernel is still Linux, so these values still work. You can see in the documentation that arg0 (the X0 register) takes an int error code for the exit syscall. By leveraging our knowledge of Linux and error codes, we can make sure that the X0 register is loaded with the result of our addition, so that we can get it out by checking its error status. There is a more powerful way to write things out to the terminal and we will look at that now by printing “Hello World” to the terminal. The below code will be saved in a file called sayHello.s

```
.global _start

.section .text
```

```
_start:
    MOV X0, 1
    LDR X1, =message
    MOV X2, 12
    MOV X8, 64
    SVC 0
    MOV X0, 0
    MOV X8, 93
    SVC 0

.section .rodata
    message:
        .ascii "Hello World\n"
```

The first new part of our program is .section .rodata This tells the linker to expect read-only data declared in this section. Moving to our start method we see lots of moves to various different registers, including a new syscall, 64.

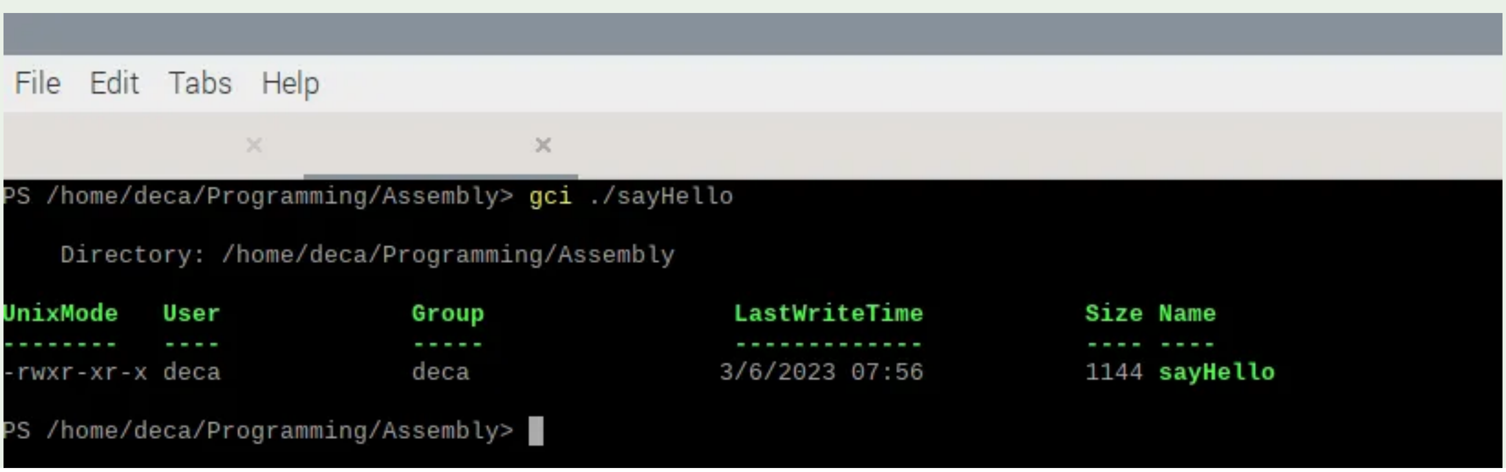
If you scroll down to row 64 in the ChromiumOS table you'll see the documentation for write. Write takes an unsigned int fd, const char *buf, and size_t count. There are three variables which mean we will need three registers to act as arguments. This means we will need to use X0, X1, and X2 to store the information we need. CPUs only understand numbers so unsigned int fd, const char *buf, and size_t count are just numbers that we need to load into the appropriate registers. The context of the write syscall is what gives them value. int fd means that the first argument is a file descriptor. A file descriptor is how Linux handles **I**nput and **O**utput operations (I/O) of which write is one of them. We need to know how Linux defines file descriptors to know how to handle this argument.

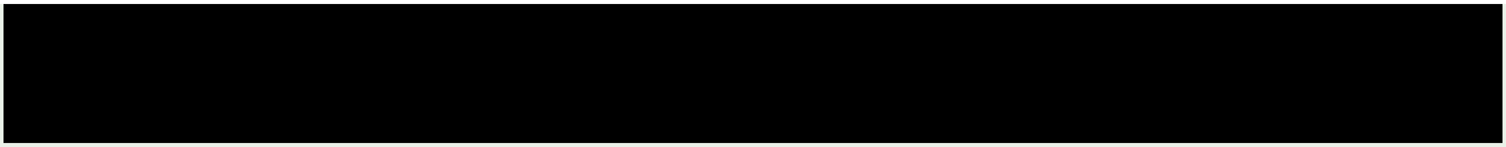
There are three default file streams in Linux called Standard Input (STDIN) assigned to the integer 0, Standard Output (STDOUT) assigned to the integer 1, and Standard Error (STDERR) assigned to the integer 2. Since write is an output it makes sense that we would need to use STDOUT (1). So, we put 1 into register X0 (argument 0). const char *buf is the next argument. The const keyword means that the value isn't modify by the program. This is why we put it in the .rodata section instead of the more common .data section. We could have just as easily put it in the .data section, but by putting it in the .rodata section, we've given more information to other programmers, as well as our future selves about how we expect the data to be used.

char *buf just means a character pointer called buf. When a variable is declared as a pointer, that just means instead of holding a value like 10, it holds the address in memory that the value 10 is stored. LDR X1, =message (a load instruction) is our way of doing this in assembly. While the syntax for LDR is similar to MOV it actually has a secret. LDR is a psuedo-instruction. A psuedo-instruction is an instruction provided by an assembler, that is then translated into a set of one or more instructions the CPU can understand. Our assembler GNU as (as or gas) contains many helpful psuedo-instructions that make coding in assembly easier[4] So even at the lowest level we still have some useful abstractions to make the code easier to write. Ultimately this line is just saying load into register X1 the memory address to the message label.

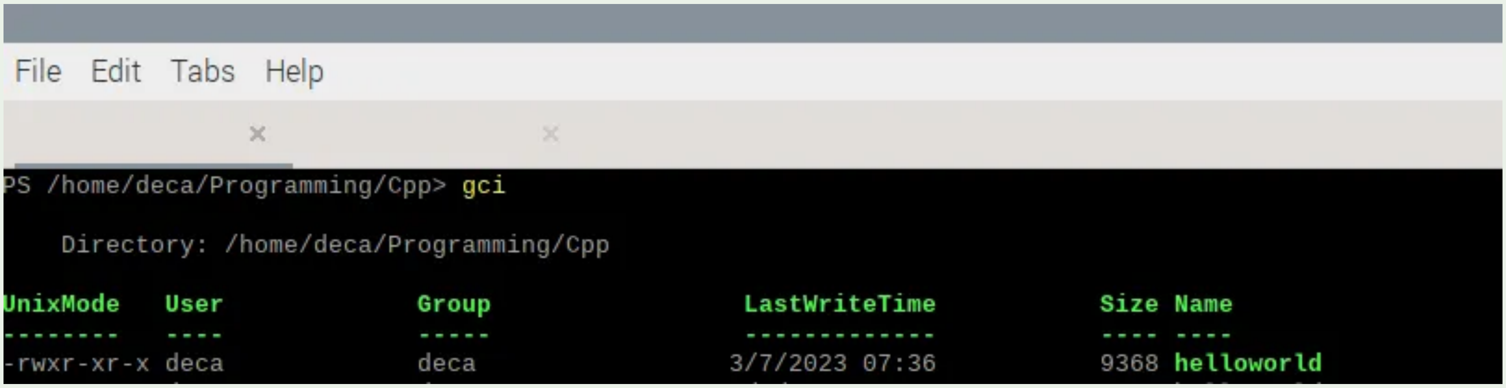
Finally, we get to our last set of registers. The value of size_t count is just a non negative number. In our case it's 12 because that is the length of our "Hello World\n" message. In our syscall register X8, we Mov the decimal value 64 (or 0x40) in hex into it. We know 64 is the correct value to use because of our syscall table. After we execute all our code it's good practice to let Linux know that we completed our program successfully. So, we manually load 0 into the X0 register before calling our exit syscall.

And that is Hello World in Arm64 assembly. If we look at the size of our sayHello program it straight dunks on a simple C version in terms of file size.





Hello World in assembly



Hello world in C

1144 bytes vs 9368. I know which one I'd take. This isn't an entirely fair comparison. C is using the standard library which is causing some code **bloat**. That's a weird sentence to say out loud. We normally think of programs like Python or JavaScript as causing overhead, but in fact C has some too. It's one of the reasons why lots of old computers need to write programs in assembly instead of C. That being said, most people would consider that to be a fair trade off.

There are some downsides to using assembly, it's not portable to other operating systems, it takes more lines of code, and it can be difficult to keep track of the flow of the program. But it's certainly a lot easier to understand when you write it from scratch, then trying to understand the assembly from a compiler afterward in Godbolt [5]

Let's up the complexity one more time in a program called compare.s

```
.EQU SYS_WRITE, 64
.EQU SYS_EXIT, 93

.global _start

_start:
        MOV X0, #5
        MOV X1, #4
        ADD X2, X0, X1
```

```
        CMP X2, #10
        BEQ equals_ten
        // If the value in register 2 does not equal 10, branch to
not_equals_10
        B not_equals_ten

not_equals_ten:
        // Write "The value is not equal to 10" to standard out
        MOV X0, #1
        LDR X1, =not_equal_message
        MOV X2, #28
        BL write_message
        B exit_program

equals_ten:
        // Write "The value equals ten" to standard out
        MOV X0, #1
        LDR X1, =equal_message
        MOV X2, #24
        BL write_message
        B exit_program

write_message:
    // Write the message to standard out
        MOV X8, SYS_WRITE
        SVC #0
        RET

exit_program:
    // Exit the program
        MOV X0, #0
        MOV X8, SYS_EXIT
        SVC #0

.section .rodata
        equal_message:
            .ascii "The value is equal to 10"
        not_equal_message:
        .ascii "The value is not equal to 10"
```

If you compile and run this program, you should get the output

```
The value is not equal to 10
```

Let's start from the top. .equ stands for equivalent. It allows us to associate numbers to a name which provides a convenient abstraction when writing assembly. Now instead of remembering magic numbers like 93, or 0x40, you can just write SYS_EXIT, or SYS_WRITE. CMP is our first new instruction, and it does what you think it does. It compares the value in a register to a number or another register. In this example we want to know if the value in register X2 is equal to 10 or not. CMP sets the status of other registers based on the result of comparing the two values. The BEQ (Branch if Equal) instruction then looks at the value of the registers set by CMP. If they are equal, it jumps to the next memory address specified by our label equals_ten if not the program continues to the next instruction. B will always jump to the next label regardless, so it acts as a useful else clause to send us to the not_equals_ten part of our program since BEQ fails. Just like we labelled _start as the entry point to our program, we can label other sections of our program to be jumped to if certain conditions are met.

In the not_equals_ten section of our program, we set up our registers to write out to the terminal the message "The value is not equal to 10" just like in our hello world example. I've moved our write call to its own label. That way I save having to write the SVC and MOV instruction twice, once for my not_equals_ten branch, and once for my equals_ten branch. It also gives me an excuse to use our next command BL. BL stands for Branch with Link. It branches just like the other branch instructions, but also saves the next instruction in the link register. In this case the instruction B exit_program is stored for later. Once we are done with our write_message: section of the code, we can return back to not_equals_ten: branch with the RET instruction. This pops the address of our branch instruction off the link register and sets the Program Counter back to our next instruction. We then execute B exit_program which takes us home with a successful exit syscall.

And that's it. That's the basics of arm assembly, registers, and syscalls. There are obviously still more instructions to learn, but you should now know enough to know where to look. With the power of assembly you can write any program that a higher level language can, though it will take more time and energy. You could even write an entire operating system like Microsoft did with MS-DOS[6]

There is one more level we could go down, which would look at how the binary instructions are deconstructed so they can be decoded. If this article generates enough interest, I might make a post about that as well 😊

Call To Action 📣

If you made it this far thanks for reading! If you are new welcome! I’ve recently started a [Twitter]() and would love for you to check it out. If you liked the article, consider liking and subscribing. And if you haven’t why not check out another article of mine! Thank you for your valuable time.

---

1 Getting work done with PowerShell on Linux (deusinmachina.net)

2 linux/unistd.h at v6.2 · torvalds/linux (github.com)

3 ChromiumOS Docs - Linux System Call Table (googlesource.com).

4 Pseudo Ops (Using as) (sourceware.org)

5 https://godbolt.org/

6 MS-DOS/v1.25/source at master · microsoft/MS-DOS (github.com)

Comments

Write a comment...