Supplement: First Assembly Program

Leonard König, November 5, 2020

Program Startup

You can think of writing a program as writing a manual for a (very) dumb machine. This manual consists of instructions which have to be executed in a specific order to reach the intended goal, but most critically, this instruction 'manual' must have first instruction, the so-called *entry point*.

In many languages, this actual programs entry point is rather hidden. If you have experience in writing Haskell programs and loading them through the GHCI, for example, then you will never have written that entry point yourself, but GHCI is providing it for you with methods to 'insert' your code or function into the already-running program. That way, your functions only need to implement the 'actual' algorithm and not worry about somehow dealing with user input or printing the result to the console—GHCI will do all that for you. You still do provide an entry point: The entry point of your function (in this exercise called `gauss`). Conversely, the Operating System provides an entry point for the BIOS of the computer to boot it: In the end our system is simply a huge chain of such 'entry points'.

The reason we do that is that this way you don't need to worry about input-output (I/O) and focus on the actual task at hand. However we do not have such a powerful tool as GHCI, instead we provide 'wrappers'. While these are written in C, a compiler can compile these into machine code. This is then combined with your code to provide a fully functioning program—both, your code and our wrapper aren't of any use alone.

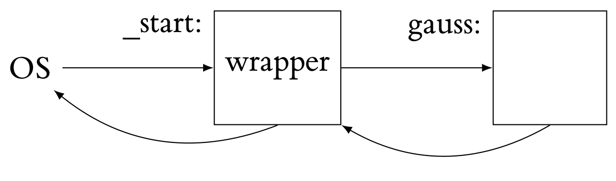

Figure 1 Wrapper provides entry point and chain-calls your code

Compilation & Linking

Our computer only accepts our ‘manual’ if the instructions given are in its native language, the machine language. Different CPUs may have different machine languages, defined by their Instruction Set Architecture (ISA). You can consider the ISA as dictionary of words (instructions) understood by the CPU, documented with their meaning.

The instructions are fed to the CPU from the memory in the format of bytes. You won’t need to learn this *machine language*; but to give a visualisation how this looks, here are a three instructions for the x86 ISA, in their binary representation:

```
0101 0101
0101 0011
1100 0011
```

Not really spectacular, but not really helpful to us either.

To ease writing programs, programmers have devised other representations of those instructions that are easier for humans to understand. These are called mnemonics and they make up an Assembly language. As the original instructions are ISA specific, their corresponding Assembly instructions must as well, since they are 1:1 translations. Moreover, there may be different Assembly ‘dialects’ translating to the same machine code. The above could be written in NASM (which is what we will use in this course) like this:

```
push rbp
push rbx
ret
```

Or in a different dialect, GAS, like this:

```
push %rbp
push %rbx
retq
```

We can now start writing the 'Gauss' program from the exercise. And while our programs entry point is provided by the wrapper, the wrapper itself requires your code to start at 'label' with the name `gauss` in order to work. Such a label is set in Assembly by simply writing its name, followed by a colon, before the first instruction that should be considered part of it.[1]

[1] The label only provides start, there's no syntactical way to denote the end. Instead we rely on assembly instructions to explicitly go back ('return').

Our first implementation of the program won't be 'correct' by any means, we simply 'return' immediately, that is, our code 'does nothing'.

```
; The following line is needed in order for our C
; code to be able to see the gauss label "globally".
global gauss

; label which works as "start-of-function" marker
; and just one instruction to immediately "return"
; from the function call.
gauss:  ret
```

Unfortunately, if the computer can't understand Assembly, we need a translator to convert the Assembly code into machine code: The Assembler. We write our code into a plain text file [2], and invoke a command line tool on it (the `$` is not typed into the console, it only signifies that the following is console input):

[2] The extension is not relevant, but you may use `.asm` or `.nasm`; likewise, the file name is irrelevant.

```
$ nasm -f elf64 -o my_code.o my_code.asm
```

This instructs the NASM assembler to produce output for our operating system (x64 Linux: ELF64), write the output into a file called `my_code.o`, and with the input code being `my_code.asm`. Files containing machine code are often called object files, hence we chose the `.o` file extension.

As we already established, this isn't a complete program yet—it still misses the wrapper. To be able to glue both parts together, we must translate it into machine code as well. The wrapper isn't written in Assembly but C, the translator—compiler—must be a different one as well:

```
$ c99 -O2 -c -o gauss_wrapper.o gauss_wrapper.c
```

The option `-O2` tells the compiler to optimize the result for speed (we require you to use this flag for different reasons!). The following option switches the compiler into 'compile' mode as the usual C compiler can actually do more than that. The option with its argument `-o gauss_wrapper.o` works the same way as for NASM.

Now we've produced the files `gauss.o` and `gauss_wrapper.o` from their respective source files. In order to create an executable in a process called 'linking', we again use the C compiler—without the `-c` option however!

```
$ c99 -o gauss gauss_wrapper.o gauss.o
```

Without the 'compile-only option' the compiler links both files together into an output we chose to call `gauss`. The name, again, is irrelevant, while on Windows systems `.exe` is used as extension, the extension doesn't matter on Linux and UNIX.

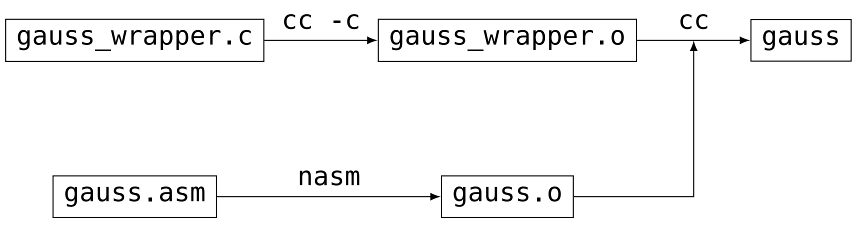

Figure 2

The last step is executing our program from the command line, an example output is shown here:

```
$ ./gauss
Not enough arguments!
$ ./gauss 2
gauss(2) = 140721287320512[WRONG]
```

We can see that despite our function returning 'nothing' the wrapper prints a result, which might be suprising at first.

x86 ISA

In order to extend our program, we need to understand some basics about x86—although most of it applies to many other architectures, just with different names and slighly changed semantics.

The most basic concept when programming Assembly are 'registers'. Our x86 CPU has quite a number of them and what they do is store 64 bit numbers.[3] For different, mostly historic, reasons they are somewhat crudely named on x86, but you can look them up any time.

[3] There are differently sized registers as well, but we will have a look at them a different time.

rax	rdi	r8	r12	(rip)
rbx	rsi	r9	r13	(rflags)
rcx	rsp	r10	r14	
rdx	rbp	r11	r15	

Table 1 x86 (AMD64) 64 bit Registers

(Almost) everything we do simply manipulates these registers contents: We can add two values in two registers, copy values from one to another, overwrite values with fixed constants, etc.

For now we will only look at arithmetics, as that's all we need for the Gauss sum. Let's try to modify our program like this:

```
global gauss

gauss:  mov rax, 30     ; rax := 30
        mov rcx, 12     ; rcx := 12
        add rax, rcx    ; rax := rax + rcx
        ret
```

This will instruct the computer to add 30 + 12 and store the result in `rax` and then return the execution from our function back to our wrapper.

We now need to run the assembler again, to produce a new `gauss.o` and then run the compiler to produce the new `gauss` executable. We can test:

```
$ ./gauss 2
gauss(2) = 42[WRONG]
```

While this is unsurpisingly still wrong, e can now understand better what the wrapper was printing in the first version of our program: The value of the register `rax` which just happened to be 140721287320512 in my case. If we can control the value of `rax`, we can control what our wrapper will print!

Calling Convention

There's no 'real' reason (as far as we are concerned) why our wrapper assumes the result to be in the register `rax`, it's a convention. This convention was developed by different operating systems vendors for the x86 architecture and as part of the clumsily named System V AMD64 ABI. There are other conventions that could be used in place of this one, but we will adhere to this *Calling Convention* everywhere, unless specified differently. This is also the place where we need to look if we want to not only output the same result all the time, but make it depend on the input.

The exercise specifies that our wrapper will take the input the user gives the program on the console, and in turn pass it to our function as its first (and only) argument. A quick look in the Calling Convention tells us that we can expect our input to be in the register `rdi`. But the contents of all other registers are unknown to us! With this knowledge, we can modify our program again, and instead of adding 12 to 30, we add whatever the user gave our program:

```
global gauss

; 1st argument: rdi
; return value: rax
gauss:  mov rax, 30     ; rax := 30
        add rax, rdi    ; rax := rax + rdi
        ret
```

Try and see for yourself!

1st	2nd	3rd	4th	5th	6th	7th,8th,...	return
rdi	rsi	rdx	rcx	r8	r9	stack	rax

Table 2 System V AMD64 CC: Parameters

This has a simple effect: Not all registers are created equal. If we'd replace `rdi` with `rsi` in the last example, the output would (likely) be vastly different.[4] The Calling Convention also has a few other restrictions for us: We aren't simply allowed to use every register as freely as we might want. Specifically, the convention defines so-called 'volatile' registers: These are the ones we are free to change within our function as *our wrapper can't expect those to be the same after our code ran*. However, our wrapper expects the following, non-volatile, registers to hold the same value after your function ran as was before: `rbx`, `rsp`, `rbp`, and `r12–r15`. Luckily we didn't use any of these yet, so we just keep it that way, for now.

[4] We could be 'lucky' and happen to be in the situation where for some reason `rsi` has the same value as `rsi`.

rax	rdi	r8	🟥 r12	🟥 (rip)
🟥 rbx	rsi	r9	🟥 r13	🟥 (rflags)
rcx	🟥 rsp	r10	🟥 r14	
rdx	🟥 rbp	r11	🟥 r15	

Table 3 System V AMD64 CC: Volatile/Non-Volatile

Furthermore, you can't directly access the registers `rip` (instruction pointer) and `rflags` (flags register), this is *technically* impossible due to the restrictions of x86. Similarly, other registers serve another special purpose as well, which we will come back to as needed.

Short x86 ISA Reference

For the exercise at hand you only need a limited set of the thousands of instructions available. While you've already learned of the instructions `mov`, `add` and `ret`, some are a bit more more complicated in their semantics. This is the case for the `mul` and `div` instructions: Both only accept one operand, despite a multiplication with just one multiplicand being quite... uninteresting.

These instructions—for reasons that do not matter here—have pre-specified arguments. That is, if you want to calculate $6 \cdot 7$, you must do this like this:

```
mov rax, 6
mov rcx, 7
mul rcx     ; rdx:rax := rax * rcx
```

The `mul` instruction multiplies its operand register with the implied register `rax`. Furthermore, as multiplications of two 64 bit numbers can easily get quite large, the result is stored as 128 bit number. We do not have such big registers, so we need to store the upper half and lower half separately, in the registers `rdx` and `rax` respectively. This means, if the result is small enough to fit into `rax` alone, `rdx` has the value 0 as it only consists of leading zeroes.[5]

[5] This simply boils down to 00031415 and 31415 being the same number. The way our processor works simply forces us to use 128 binary digits.

The division conversely divides the 128 bit number consisting of the concatenation of `rdx:rax` by its operand. Thus, to divide $126 \div 3$, you need to store the dividend in it's 128 bit represenation, with the upper half filling `rdx`. As $126 < 2^{64}$, the upper half will be simply zero.

```
mov rax, 126
mov rdx, 0      ; 64 leading zeroes
mov rcx, 3
div rcx         ; rax := rdx:rax ÷ rcx
```

In contrast to maths however, if we forget to clear `rdx` to zero, it's (likely non-zero) contents will be seen as the upper half of our dividend, resulting in quite a different result.