Richard JP Le Guen.ca

Skip to Content
All about Software Development on the WWW

- Richard JP Le Guen
- Curriculum Vitae
- Tutoring

Search
RSS feed

Navigation

- Home
- Curriculum Vitae
- Tutoring
- Old News

Tutoring Courses

- SOEN 229: System Software
- SOEN 287: Introduction to Web Applications (fall 2009)
- SOEN343: Software Design and Architecture (fall 2010)

- Richard JP Le Guen.ca
- Old News
- An Intro To NASM Programming - For SOEN228

An Intro to NASM Programming - for SOEN228 March 14th, 2011

I never really learned to use NASM to its full potential, and now – in SOEN228, Concordia's System Hardware class which I'm currently tutoring – I'm supposed to be teaching students how to use it.

To that end, I've written up a little refresher for me to consult.

Please note that – in order to discourage plagiarizers – all quotation marks – both "double quotes" and 'single quotes' – in this article (and all pages on my web site) are not normal quotation marks. If you are copy-pasting code it won't compile or run properly.

The Plan: A "Hello World!" written in the NASM Assembly Language

What I'm going to do is to write a Hello World program, one step at a time… very slowly. Changes will be made one at a time, resulting in several iterations of the Hello World program. Some of them will run and compile without problem; some of them will fail miserably. Eventually, it will evolve to prompt the user for input ("What is your name?") so as to output personalized Hello messages such as "Hello Richard".

Before I get into it, if you are a student of SOEN228, you will also want to read the tutorials I wrote on ViM and SSH for SOEN229 – both tools you will need to understand before moving on to the next semester of the Software Engineering program, and you should be using them during these SOEN228 labs. Once you're familiar with ViM and SSH, connect to one of Concordia's public access servers and create a `HelloWorld.asm` file like so: (note that the `[prompt] >` is the prompt; don't type that part out)

```
[prompt] > vim HelloWorld.asm
```

The Program – Iteration 1

Before I write the program, I need to reflect on what that program is doing. If I were programming at a higher level of abstraction (such as in Java or in C) I could probably summarize the program as simply "outputting the String 'Hello World!'" but – when writing in Assembly language – this is an oversimplification which won't work. That description of the Hello World program neglects that the String "Hello World!" (really not a string but simply information) must be declared and stored somewhere in memory.

So, again, in summary the program must at least do the following: (this list will grow)

1. Declare/store the information "Hello World!" (data)
2. Output that information "Hello World!" (behavior)

Declare/store the information "Hello World!" (data)

In NASM Assembly language, a Data Segment is used to declare data like this. Use `db` (which stands for "declare byte"; ignore the "byte" part and just think about the word "declare" for now…)

```
segment .data
        helloMsg db 'Hello World!'

        ; more to come…
```

Output that information "Hello World!" (behavior)

The behavior of my program consists of instructions and I'm going to put code executing instructions in the Text Segment of my program.

Text Segment

The Text Segment must include a label `_start` – think of it as the NASM equivalent of `main(…)` in Java. That label (`_start`) must be declared as global… just because.

```
segment .data
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                ; more to come…
```

Executing Instructions

I'm now ready to actually *do something* with my program. What I need to do is output the information at `helloMsg`… but how do I do that?

Sadly, there is no instruction for output in NASM assembly language programming; I'm going to have to use a System Call. Invoking a system call is done by invoking an interrupt with the `int` instruction:

```
segment .data
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                ; SOMETHING IS MISSING HERE
                int 0 × 80      ; invoke an interrupt
                ; more to come…
```

If you're a SOEN228 student, take note that whenever you invoke a system call you will use `int 0 × 80`; within the scope of this course, all interrupts will always be `int 0 × 80` interrupts – you don't need to think about the other interrupts in SOEN228.

But there are a lot of different system calls; exit, read and write are all system calls… and many more. How does the system know whether my program is trying to invoke read or write, or even exit?

Each system call is (arbitrarily) numbered, and when the `int 0 × 80` interrupt is invoked (interrupting the execution of the program and transferring control to the operating system kernel so as to execute the system call) the operating system looks at the value in the eax register to identify which system call should be executed. The System Call Table I linked to above lists the numeric value for each system call in a column labeled "%eax".

The system call I need to perform an output is `sys_write`; in effect, when my program outputs it's really writing to screen. When I look it up in the Linux System Call Table or the Linux System Call Quick Reference, I can see that `sys_write` is system call 4.

```
segment .data
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                mov eax, 4
                ; SOMETHING IS MISSING HERE
                int 0 × 80      ; invoke an interrupt
                ; more to come…
```

I'm not done though; I also need to specify *where* the operating system should write to (should it write to a file? to speakers?) and *what* it has to write. Those are done with ebx and ecx respectively:

```
segment .data
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                mov eax, 4
                mov ebx, 1
                mov ecx, helloMsg
                ; SOMETHING IS *STILL* MISSING HERE
                int 0 × 80      ; invoke an interrupt
                ; more to come…
```

It's important to note that `helloMsg` is not the string 'Hello World!' but rather the string's address in memory.

To emphasize this, consider the size of the ecx register versus the size of 'Hello World!'. 'Hello World!' is 12 characters long; each character in ASCII is 1 byte. Therefore 'Hello World!' would be 12 bytes long, while ecx is only 4 bytes long. As such, the information 'Hello World!' can't be moved into ecx; it's just not possible.

Which brings me to the last piece of missing information before I invoke the system call with the interrupt `0 × 80`: ecx holds the address of the beginning of the information, but since `helloMsg` is simply an address it has no implicit length. This isn't Java; 'Hello World!' isn't a String object with a handy-dandy `length` property. Since `helloMsg` is merely an address, I need to specify how many bytes I'm writing to screen, using edx:

```
segment .data
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 12     ; 12 bytes (characters) of that information…
                int 0 × 80      ; invoke an interrupt
                ; more to come…
```

Compiling and Running the Program

I'm going to take a break here because it looks like I'm ready to compile and run the program. I was expecting my program to:

1. Declare/store the information "Hello World!" (data)
2. Output that information "Hello World!" (behavior)

... and it looks like it's doing that.

```
segment .data
        ; Declare/store the information "Hello World!"
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                ; Output that information "Hello World!"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 12     ; 12 bytes (characters) of that information…
                int 0 × 80      ; invoke an interrupt
                ; more to come?
```

To compile my program, on the Linux command line, I use the following instruction: (again, `[prompt] >` is the prompt; don't type that!)

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
```

For reference purposes, `nasm` is an assembler and `ld` is a linker – but that information is technically out of scope for SOEN228.

Now, to run my program I type the following on the command line:

```
[prompt] > ./HelloWorld
```

... and I get the following output:

```
HelloWorld!Segmentation fault
```

My First Segmentation Fault

A Segmentation Fault is a memory access violation… but it doesn't look like there should be a segmentation fault in my program – it's a simple Hello World, isn't it?

Well, I forgot something very important. In order to understand this, I have to think about some computer architecture for a moment: specifically, I'm thinking about the Program Counter, the MAR and the MDR – some major players in the Instruction Cycle.

Simply put, to execute each instruction the value in the Program Counter register is moved to the MAR, then the information (in memory) at that address is read into the MDR, and that information is executed as an instruction. Meanwhile, the Program Counter increments by one so as to point to the next instruction.

So consider what happens when the program execution reaches that last line (`int 0 × 80`): the value in the Program Counter is moved into the MAR and the data at that address is memory is read into memory so it can be executed as an instruction… *but there is no next instruction*. Therefore the next position in memory isn't the next instruction but some memory which doesn't belong to my program; hence, a segmentation fault.

This brings me to Important NASM Lesson #1.

Important Lesson #1: Always Make A System Call to `sys_exit`

A NASM program needs to explicitly exit; I can't just let execution end like I would in higher abstraction languages. So really, what I need the program to do (continuation of the list from above) is:

1. Declare/store the information "Hello World!" (data)
2. Output that information "Hello World!" (behavior)
3. Exit

At the end of a NASM program I will always have to make a system call to `sys_exit`, which is system call 1. So I need to add an interrupt invoking system call 1 to the end of my text segment:

```
segment .data
        ; Declare/store the information "Hello World!"
        helloMsg db 'Hello World!'

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                ; Output that information "Hello World!"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
```

```
                    mov edx, 12     ; 12 bytes (characters) of that information…
                    int 0 × 80      ; invoke an interrupt
                    ; Exit
                    mov eax, 1      ; sys_exit
                    mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                    ; see http://en.wikipedia.org/wiki/Exit_status
                    int 0 × 80
```

Compiling and Running the Program

So I'm ready to compile and run again.

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
```

This time, I run the program and get the following output:

```
Hello World!
```

But there's something that bothers me: the next prompt is on the same line as my output:

```
[prompt] > ./HelloWorld
Hello World![prompt] >
```

...which is ugly. So, my next iteration of the Hello World program will be just like this one, but will include that new line between the end of the output and the next prompt.

The Program – Iteration 2

How can I add a new line to the end of my output? I'm going to consider some (wrong) choices:

Can I Just Put a New Line in my Data Segment?

What if I change my data segment – adding a new line in the "Hello World!" data, so it spans two lines?

```
helloMsg db 'Hello World!
'       ; this won't work!
```

This won't work, and why seems kind of apparent to me when I consider the difference between semi-colons in other languages and in NASM. In NASM semicolons are used to create line comments, but in other languages it is used as a statement terminator. In NASM, instead, the new line character is used as a statement terminator.

As such if I use the above code and try to assemble with `nasm` on the command line, I get an error:

```
[prompt] > nasm -f elf HelloWorld.asm
HelloWorld.asm:3: warning: unterminated string
HelloWorld.asm:3: error: expression syntax error
HelloWorld.asm:4: warning: unterminated string
HelloWorld.asm:4: error: label or instruction expected at start of line
```

Can I Use an Escape Character like `\n`?

In Java, a higher-abstraction language than NASM, I could use an esacpe character `\n` to include a new line in a string, but this is not the case in NASM:

```
helloMsg db "Hello World!\n"    ; this won't work!
```

This is not right. To my knowledge, there are no such escape characters in NASM. Using the above I could end up with an application which outputs the following:

```
[prompt] > ./HelloWorld
```

```
Hello World!\n[prompt] >
```

No improvement.

Can I Use an ASCII code?

Using an escape character was in the right ballpark, but the correct solution is to use an ASCII code. I can't put the ASCII code in quotes though:

```
helloMsg db "Hello World!", 0xA ; 0xA is the ASCII code for new-line
```

So what if I compile and run again? My program still doesn't work!

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
[prompt] > ./HelloWorld
Hello World![prompt] >
```

The problem is I changed the length of my output "string" – the information located at address `helloMsg` – but I didn't change the length of the output – the edx register when I invoke the system call.

So, for the program to work properly I need to change the value in edx from 12 to 13:

```
segment .data
        ; Declare/store the information "Hello World!"
        helloMsg db 'Hello World!',0xA

segment .text

        global _start   ; define the label _start as global

        _start: ; the _start label

                ; Output that information "Hello World!"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 13     ; 13 bytes (characters) of that information…
                int 0 × 80      ; invoke an interrupt
                ; Exit
                mov eax, 1      ; sys_exit
                mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                ; see http://en.wikipedia.org/wiki/Exit_status
                int 0 × 80
```

I compile and run again…

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
[prompt] > ./HelloWorld
Hello World!
[prompt] >
```

... and now I'm ready to move on to the next iteration of my Hello World program.

The Program – Iteration 3

Now I'm going to change the program so it accepts some user input. I want it to behave as follows:

1. Prompt the user for their name
2. Accept input
3. Display `Hello {name}!`

Since this is NASM assembly programming, though, I've taken a few things for granted: (again, this list will grow)

1. Declare/store the information `"What is your name? "` (data)
2. Declare/store the information `"Hello "` (data)
3. Declare/store the information `"!",0xA` (data)
4. Prompt the user for their name (behavior)
5. Accept input and store the user's name (behavior? data?)
6. Display `Hello {name}!`, or: (more specifically)
    1. Display `"Hello "` (behavior)
    2. Display the user's name (behavior)
    3. Display `"!",0xA` (behavior)
7. Exit (behavior)

Declare/store the information… (data)

I need to store the string of characters `"What is your name"` as well change `"Hello World!",0xA` to two separate strings of characters `"Hello "` and `"!",0xA` so I'm going to add it to my Data Segment:

```
segment .data
        ; Declare/store the information "Hello World!"
        prompt db 'What is your name? '
        helloMsg db 'Hello '
        endOfLine db '!',0xA

        segment .text
        ; omitted…
```

I'm also going to declare data in my Data Segment for the user's name. **This is 100% wrong** and I'll get to that soon, but for now I'm experimenting with the Data Segment for the sake of demonstration. **It is of utmost importance that the name is placed between `helloMsg` and `endOfLine`.**

Since the user's name has no value to begin with (one could call it **uninitialized**) I'm initializing it with a value of 8 spaces:

```
segment .data
        ; Declare/store the information "Hello World!"
        prompt db 'What is your name? '
        ; do not change the order of the following three lines!
        helloMsg db 'Hello '
        name db '        '      ; 8 space characters
        endOfLine db '!',0xA
        ; do not change the order of the previous three lines!

        segment .text
        ; omitted…
```

Prompt the user for their name (behavior)

Since I'm into the behavior of my program, I'm no longer working in `segment .data` but rather in `segment .text`.

When you boil down to it, prompting is really just output – so that means a system call to `sys_write`. Where there's a system call, there's an interrupt:

```
                ; Output that information 'What is your name? '
                ; more to come…
                int 0 × 80
```

Again, in order to distinguish between the different system calls, I have to use eax in conjunction with the arbitrary system call number for `sys_write` – which is 4:

```
                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
```

```
; more to come…
int 0 × 80      ; invoke an interrupt
```

I have to specify where I'm writing to with ebx:

```
; Output that information 'What is your name? '
mov eax, 4      ; write…
mov ebx, 1      ; to the standard output (screen/console)…
; more to come…
int 0 × 80      ; invoke an interrupt
```

And I again have to specify what I'm outputting (in memory) and how much of it:

```
; Output that information 'What is your name? '
mov eax, 4      ; write…
mov ebx, 1      ; to the standard output (screen/console)…
mov ecx, prompt ; the information at memory address prompt
mov edx, 19     ; 19 bytes (characters) of that information
int 0 × 80      ; invoke an interrupt
```

Accept input and store the user's name (behavior)

Input is – like output – not one of the instructions available to me when programming NASM. As such, I again need to make another system call, which means invoking an interrupt:

```
; Accept input and store the user's name
; more to come…
int 0 × 80      ; always 0 × 80
```

The eax register is used to distinguish between output and input system calls. I have to set its value (using a `mov` instruction) to the numerical value arbitrarily assigned to the `sys_read` (input) system call. Checking a Linux System Call Table I can see that the numeric value expected in eax for `sys_read` is 3.

```
; Accept input and store the user's name
mov eax, 3      ; read…
; more to come…
int 0 × 80
```

Just like with output, I need to identify an input device (am I accepting input from the mouse? the ethernet cable?) using ebx:

```
; Accept input and store the user's name
mov eax, 3      ; read…
mov ebx, 1      ; from the standard input (keyboard/console)…
; more to come…
int 0 × 80
```

And, again just like with input, I need to specify where I'm storing the input in memory using ecx:

```
; Accept input and store the user's name
mov eax, 3      ; read…
mov ebx, 1      ; from the standard input (keyboard/console)…
mov ecx, name   ; storing the information at name
; more to come…
int 0 × 80
```

... and I have to use edx to a specify the size of what I'm writing in memory:

```
; Accept input and store the user's name
mov eax, 3      ; read…
mov ebx, 1      ; from the standard input (keyboard/console)…
mov ecx, name   ; storing at memory location name…
mov edx, 8      ; 8 bytes (characters) of that information
int 0 × 80
```

Display `Hello {name}!` (behavior)

I need to change my output from before (previously `"Hello World!",0xA`) to now output `"Hello"` then `name` then `"!",0xA`, but I'm going to use a little trick to do it: instead of having 3 different outputs (an system calls) I'm going to have one:

```
                ; Output that information "Hello…"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 16     ; 16 bytes (characters) of that information…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80
```

... and I've specified the length (in edx) as 16 bytes. (why will become clear when I run the program)

Exit

I already used the `sys_exit` system call in an earlier iterations of the Hello World program.

```
                ; exit
                mov eax, 1      ; sys_exit
                mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                ; see http://en.wikipedia.org/wiki/Exit_status
                int 0 × 80
```

The Whole Program

So altogether, my program currently should look something like this:

```
segment .data
        ; Declare/store the information "Hello World!"
        prompt db 'What is your name? '
        ; do not change the order of the following three lines!
        helloMsg db 'Hello '
        name db '        '      ; 8 space characters
        endOfLine db '!',0xA
        ; do not change the order of the previous three lines!

segment .text

        global _start

        _start:

                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, prompt ; the information at memory address prompt
                mov edx, 19     ; 19 bytes (characters) of that information
                int 0 × 80      ; invoke an interrupt

                ; Accept input and store the user's name
                mov eax, 3      ; read…
                mov ebx, 1      ; from the standard input (keyboard/console)…
                mov ecx, name   ; storing at memory location name…
                mov edx, 8      ; 8 bytes (characters) of that information
                int 0 × 80

                ; Output that information "Hello…"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 16     ; 16 bytes (characters) of that information…
```

```
                        ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80

                ; Exit
                mov eax, 1      ; sys_exit
                mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                ; see http://en.wikipedia.org/wiki/Exit_status
                int 0 × 80
```

Compiling and Running the Program

If I again compile and run the program, providing my own name ("Richard") as input, I get something like this…

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
[prompt] > ./HelloWorld
What is your name? Richard
Hello Richard
!
[prompt] >
```

Observations About Iteration 3

There are some ideas and points which stand out about Iteration 3 for me.

Important Lesson #2: The Data Segment Is Continuous

The important lesson I've picked up from Iteration 3 is that the data segment is continuous; since I declared `helloMsg`, `name` and `endOfLine` one after another, they are adjacent to each other in memory, which allows me to output them all with a single `sys_write` which outputs 16 bytes beginning at the address `helloMsg`:

```
                ; Output that information "Hello…"
                ; since the data segment is continuous
                ; and helloMsg, name and endOfLine are one after another
                ; I can just output all 16 bytes with one sys_write
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, 16     ; 16 bytes (characters) of that information…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80
```

This is just further evidence that the values declared in the data segment using `db` are not really variables nor objects nor Strings: they are simply positions in a segment of memory which have been given names. There is no concrete distinction between the 3 strings of information declared in the data segment of my program.

Calculating the length of information using `equ` and `$`

Since the data segment is continuous, I should be able to automatically determine the length of the information I output. For example, consider `prompt`:

```
segment .data
        ; Declare/store the information "Hello World!"
        prompt db 'What is your name? '
        ; do not change the order of the following three lines!
        helloMsg db 'Hello '

        ; omitted

        segment .text

        global _start
        _start:

                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, prompt ; the information at memory address prompt
                mov edx, 19     ; 19 bytes (characters) of that information
                int 0 × 80      ; invoke an interrupt

        ; omitted
```

Since `helloMsg` is the memory address at the *end* of the information 'What is your name? ' I should be able to calculate the length of 'What is your name? ' automatically by subtracting `prompt` from `helloMsg`. (end – beginning = length)

This can be done with `equ` and the `$` token. In the data segment `equ` can be used to give a symbol a constant value, instead of making it an address, while the `$` token is treated as assembly position of the current line.

So in the above code I can replace the hardcoded value `19` with a new symbol in my data segment, `promptLength`:

```
segment .data
        ; Declare/store the information "Hello World!"
        prompt db 'What is your name? '
        promptLength equ $-prompt       ; = (the current address) - (the address of prompt)
        ; do not change the order of the following three lines!
        helloMsg db 'Hello '

        ; omitted

        segment .text

        global _start

        _start:

                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, prompt ; the information at memory address prompt
                ; now I can use that new symbol, promptLength
                mov edx, promptLength   ; length of information in bytes
                int 0 × 80      ; invoke an interrupt

        ; omitted
```

Now, if I have to change the prompt from 'What is your name? ' to 'Please input your full name? ' I don't have to worry about updating (or forgetting to update) the length of the output for the `sys_write` – the assembler will re-calulate the value of `promptLength` for me.

Important Lesson #3: Using the BSS section

Earler I said that I was putting `name` in the data segment and this was wrong; this is because `name` is uninitialized data, and only initialized data goes in the data segment. Uninitialized data belongs in the BSS segment. (why? because!)

So I need to make some changes to my program so as to respect that rule: I need to remove `name` from `segment .data`, and while I'm at it I might as well make use of the above trick to calculate the length of information in memory:

```
segment .data
        prompt db 'What is your name? '
        helloMsg db 'Hello '
        helloMsgLength equ $-helloMsg
        endOfLine db '!',0xA
        endOfLineLength equ $-endOfLine
```

I also need to add a `segment .bss`. In `segment .bss` – since unintialized data goes here – I'm not *declaring* information (like I was in `segment .data`) but rather just reserving memory. Hence, in `segment .bss` I won't be using `db` but rather `resb` – which stands for "reserve byte":

```
segment .bss
        name: resb 8
```

And lastly, I have to change the call to `sys_write` which outputs `helloMsg`. (since the `name` is no longer in the data segment the information is no longer continuous in memory)

Here's the whole program once I've made those changes:

```
segment .data
        prompt db 'What is your name? '
        helloMsg db 'Hello '
        helloMsgLength equ $-helloMsg
        endOfLine db '!',0xA
        endOfLineLength equ $-endOfLine

segment .bss
        name: resb 8

segment .text

        global _start

        _start:

                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, prompt ; the information at memory address prompt
                mov edx, 19     ; 19 bytes (characters) of that information
                int 0 × 80      ; invoke an interrupt

                ; Accept input and store the user's name
                mov eax, 3      ; read…
                mov ebx, 1      ; from the standard input (keyboard/console)…
                mov ecx, name   ; storing at memory location name…
                mov edx, 8      ; 8 bytes (characters) of that information
                int 0 × 80

                ; Output that information "Hello…"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, helloMsgLength ; length of information in bytes…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80

                ; Output that information name
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, name   ; the information at name…
                mov edx, 8      ; 8 bytes (characters) of that information…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80

                ; Output that information "!",0xA
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, endOfLine      ; the information at helloMsg…
                mov edx, endOfLineLength        ; length of that information in bytes…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80
                ; Exit
                mov eax, 1      ; sys_exit
                mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                ; see http://en.wikipedia.org/wiki/Exit_status
                int 0 × 80
```

Important Lesson #4: That New Line Character…

The last important thing I'm going to take away from Iteration 3 is that there is a new line character in the output, between my name and the exclamation point '!' – which shouldn't be there:

```
What is your name? Richard
Hello Richard
!
```

In order to understand where this new line is coming from, I have to think really hard about what the user types in when prompted, because while my name is 7 characters long, *the input the user provides is not*. I have to remember that to input my name, the user has to type in 'R'-'i'-'c'-'h'-'a'-'r'-'d'-**ENTER**. That enter is a new line character, and it is included with the input provided by the user. For my Hello World program to work properly, I have to remove the new line character, and everything after it; this will be Iteration 4.

The Program – Iteration 4

In this final iteration I am going to write some code which will remove the new line character from the user input. In order to do this I need to:

1. Declare/store the information `"What is your name? "` (data)
2. Declare/store the information `"Hello "` (data)
3. Declare/store the information `"!",0xA` (data)
4. Prompt the user for their name (behavior)
5. Accept input and store the user's name (behavior? data?)
6. Display `Hello {name}!`, or: (more specifically)
   1. Display `"Hello "` (behavior)
   2. Loop over the the user's name outputting every character until a new line (behavior)
   3. Display `"!",0xA` (behavior)
7. Exit (behavior)

It should be noted that while my program will loop logically, I will not be using the `LOOP` keyword.

I've dramatically oversimplified looping over the characters in the user's name though. As en example, if I think about a `for` loop in a higher-abstraction language like Java, I realize that a for loop is actually composed of several smaller instructions:

```
for(int i=n; i<n+8; i++)
```

I have the initializer, the loop-test (condition) and the counting expression. Also, I have a variable `i` which is being initialized, compared against the conditional and changed with each loop. I am going to call this value the "index".

So really, a loop can be no less complex than:

1. Initialize the index
2. (Do something in the loop)
3. Change the value of the index
4. Perform the loop test; if a success, loop again

So my program now has to:

1. Declare/store the information `"What is your name? "` (data)

2. Declare/store the information `"Hello "` (data)
3. Declare/store the information `"!",0xA` (data)
4. Prompt the user for their name (behavior)
5. Accept input and store the user's name (behavior? data?)
6. Display `Hello {name}!`, or: (more specifically)
    1. Display `"Hello "` (behavior)
    2. Loop over the the user's name outputting every character until a new line, or: (behavior)
        1. Initialize an index
        2. Check if the current character is a new line
            - If it is, exit the loop
            - If it is not, output the character
        3. Change the value of the index
        4. Perform the loop test; if a success, loop again.
    3. Display `"!",0xA` (behavior)
7. Exit (behavior)

Up until the loop, my program's behavior is unaffected, so I am going to skip over everything before it.

Initialize an index

There are special registers which are conventionally used as indexes. They are called index registers and the ones available for me to use when I'm writing an assembly language program are esi – the source index registry – and edi – the destination index registry. The source index registry (esi) is conventionally used when reading from memory, and since I'm reading from memory and writing to the standard output this is the index I will use as my index. (realistically I could use any registry I want, but this is the conventional one to use)

I have to initialize it to the starting position, which is at `name`:

```
mov esi, name   ; initialize the source index
```

Check if the current character is a new line

In order to check if the current character is a new line, I need to perform a comparison using `cmp`. I'm trying to compare the character which is in-memory at the address currently stored in register esi. Therefore, I'm going to be using Register Indirect Addressing.

At this point, I'm tempted to try this:

```
cmp [esi], 0xA  ; this won't work
```

... but this won't work. There's nothing to indicate the size of the information being compared. Should 1 byte of memory be compared to `0xA`? 2 bytes? 3 bytes?

In order to specify the size of the comparison, I have to include a register as one of the operands in my `cmp` instruction:

```
mov esi, name   ; initialize the source index
; more to come…
mov al, [esi]   ; al is a 1 byte register
cmp al, 0xA
```

My comparison should have a consequence, however; if the comparison finds that the two values are equal, my program should exit the loop. I can accomplish this by means of a "jump if equal" instruction – `je`:

```
mov esi, name   ; initialize the source index
; more to come…
mov al, [esi]   ; al is a 1 byte register
cmp al, 0xA     ; if al holds an ASCII new line…
je exitLoop     ; then jump to label exitLoop
```

```
        ; more to come…

exitLoop:       ; the label to which the je above jumps
        ; omitted
```

If the comparison finds that the two values are not equal (the character is not a new line) then I continue and output the character:

```
        mov esi, name   ; initialize the source index
        ; more to come…
        mov al, [esi]   ; al is a 1 byte register
        cmp al, 0xA     ; if al holds an ASCII new line…
        je exitLoop     ; then jump to label exitLoop

        ; if al does not hold an ASCII new line…
        mov eax, 4      ; write…
        mov ebx, 1      ; to standard output…
        mov ecx, esi    ; the information at address esi..
        mov edx, 1      ; 1 byte (character) of that information
        int 0 × 80

        ; more to come…

exitLoop:       ; the label to which the je above jumps
        ; omitted
```

Change the value of the index

I now need to increment the index esi by 1:

```
        mov esi, name   ; initialize the source index
        ; more to come…
        mov al, [esi]   ; al is a 1 byte register
        cmp al, 0xA     ; if al holds an ASCII new line…
        je exitLoop     ; then jump to label exitLoop

        ; if al does not hold an ASCII new line…
        mov eax, 4      ; write…
        mov ebx, 1      ; to standard output…
        mov ecx, esi    ; the information at address esi..
        mov edx, 1      ; 1 byte (character) of that information
        int 0 × 80

        add esi, 1      ; you could also use the inc instruction
        ; more to come…

exitLoop:       ; the label to which the je above jumps
        ; omitted
```

Perform the loop test; if a success, loop again

Lastly, I have to ensure that once I've looped over all the characters in the input, I exit the loop:

```
        mov esi, name   ; initialize the source index
        ; more to come…
        mov al, [esi]   ; al is a 1 byte register
        cmp al, 0xA     ; if al holds an ASCII new line…
        je exitLoop     ; then jump to label exitLoop

        ; if al does not hold an ASCII new line…
        mov eax, 4      ; write…
        mov ebx, 1      ; to standard output…
        mov ecx, esi    ; the information at address esi..
        mov edx, 1      ; 1 byte (character) of that information
        int 0 × 80

        add esi, 1      ; you could also use the inc instruction

        cmp esi, name+8 ; see if esi is pointing past the end of the 8 reserved bytes
        jl loopAgain

exitLoop:       ; the label to which the je above jumps
        ; omitted
```

Here I'm comparing esi to `name+8` to see if esi is pointing past the last byte reserves for `name`. I jump if esi is less than `name+8` (using `jl` – 'jump less than') to a labal `loopAgain`… except there is no such label. I have to add it after I initialized esi:

```
        mov esi, name   ; initialize the source index

loopAgain:
        mov al, [esi]   ; al is a 1 byte register
        cmp al, 0xA     ; if al holds an ASCII new line…
        je exitLoop     ; then jump to label exitLoop

        ; if al does not hold an ASCII new line…
        mov eax, 4      ; write…
        mov ebx, 1      ; to standard output…
        mov ecx, esi    ; the information at address esi..
        mov edx, 1      ; 1 byte (character) of that information
        int 0 × 80

        add esi, 1      ; you could also use the inc instruction

        cmp esi, name+8 ; see if esi is pointing past the end of the 8 reserved bytes
        jl loopAgain

exitLoop:       ; the label to which the je above jumps
        ; omitted
```

The Final Program

```
segment .data
        prompt db 'What is your name? '
        helloMsg db 'Hello '
        helloMsgLength equ $-helloMsg
        endOfLine db '!',0xA
        endOfLineLength equ $-endOfLine

segment .bss
        name: resb 8

segment .text

        global _start

        _start:

                ; Output that information 'What is your name? '
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, prompt ; the information at memory address prompt
                mov edx, 19     ; 19 bytes (characters) of that information
                int 0 × 80      ; invoke an interrupt

                ; Accept input and store the user's name
                mov eax, 3      ; read…
                mov ebx, 1      ; from the standard input (keyboard/console)…
                mov ecx, name   ; storing at memory location name…
                mov edx, 8      ; 8 bytes (characters) of that information
                int 0 × 80
                ; Output that information "Hello…"
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, helloMsg       ; the information at helloMsg…
                mov edx, helloMsgLength ; length of information in bytes…
                                ;       (but isn't helloMsg only 6 bytes?)
                int 0 × 80

                mov esi, name   ; initialize the source index

        loopAgain:
                mov al, [esi]   ; al is a 1 byte register
                cmp al, 0xA     ; if al holds an ASCII new line…
                je exitLoop     ; then jump to label exitLoop

                ; if al does not hold an ASCII new line…
                mov eax, 4      ; write…
                mov ebx, 1      ; to standard output…
                mov ecx, esi    ; the information at address esi..
                mov edx, 1      ; 1 byte (character) of that information
                int 0 × 80

                add esi, 1      ; you could also use the inc instruction

                cmp esi, name+8 ; see if esi is pointing past the end of the 8 reserved bytes
                jl loopAgain

        exitLoop:               ; the label to which the je above jumps


                ; Output that information "!",0xA
                mov eax, 4      ; write…
                mov ebx, 1      ; to the standard output (screen/console)…
                mov ecx, endOfLine      ; the information at helloMsg…
                mov edx, endOfLineLength        ; length of information in bytes…
                int 0 × 80

                ; Exit
                mov eax, 1      ; sys_exit
                mov ebx, 0      ; exit status. 0 means "normal", while 1 means "error"
                                ; see http://en.wikipedia.org/wiki/Exit_status
                int 0 × 80
```

I compile and run, and this time I get…

```
[prompt] > nasm -f elf HelloWorld.asm
[prompt] > ld -o HelloWorld HelloWorld.o
[prompt] > ./HelloWorld
What is your name? Richard
Hello Richard!
[prompt] >
```

Yaaay!

Like and 5 others liked this.

Add New Comment

Login

Showing 6 comments

Sort by popular now

manu

thank u so much.....i googled a lot 4 a simple code,but couldn't find .this was really helpful....easy to understand.

2 months ago 1 Like Like Reply

Arun P G

Super tutorial for Nasm programming

1 month ago Like Reply

Asit

simply superb............too much helpful for me........thank u vry much sir....

2 months ago Like Reply

jimy

Very very helpful and thanks for sharing it !!

5 months ago Like Reply

David Chouinard

This is a very well written guide to NASM, thanks you for talking the time to write it.

5 months ago Like Reply

Abcd

Very Helpfullll thnkss

11 months ago Like Reply

Subscribe by email RSS

Trackback URL http://disqus.com/forums/r

blog comments powered by DISQUS

Content © 2008-2012 Richard Jean-Paul Le Guen