2-64-asm-hello-world-triangle-puts-fibo

https://aslrfellow.github.io/

https://jameshfisher.com/2018/03/10/linux-assembly-hello-world/

- The system call number is put in rax.
- Arguments are put in the registers rdi, rsi, rdx, rcx, r8 and r9, in that order.
- The system is called with the syscall instruction.
- The return value of the system call is in rax. An error is signalled by returning -errno.

https://cs.lmu.edu/~ray/notes/nasmtutorial/

```
;
-----------------------------------------------------------------------------
------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit
macOS only.
; To assemble and run:
;
;     nasm -f macho64 hello.asm && ld -macosx_version_min 10.7.0 -lSystem -o
hello hello.o && ./hello
;
-----------------------------------------------------------------------------
------

        global    start

        section   .text
start:    mov       rax, 0x02000004         ; system call for write
        mov       rdi, 1                  ; file handle 1 is stdout
        mov       rsi, message            ; address of string to output
        mov       rdx, 13                 ; number of bytes
        syscall                           ; invoke operating system to do the write
        mov       rax, 0x02000001         ; system call for exit
        xor       rdi, rdi                ; exit code 0
        syscall                           ; invoke operating system to exit
```

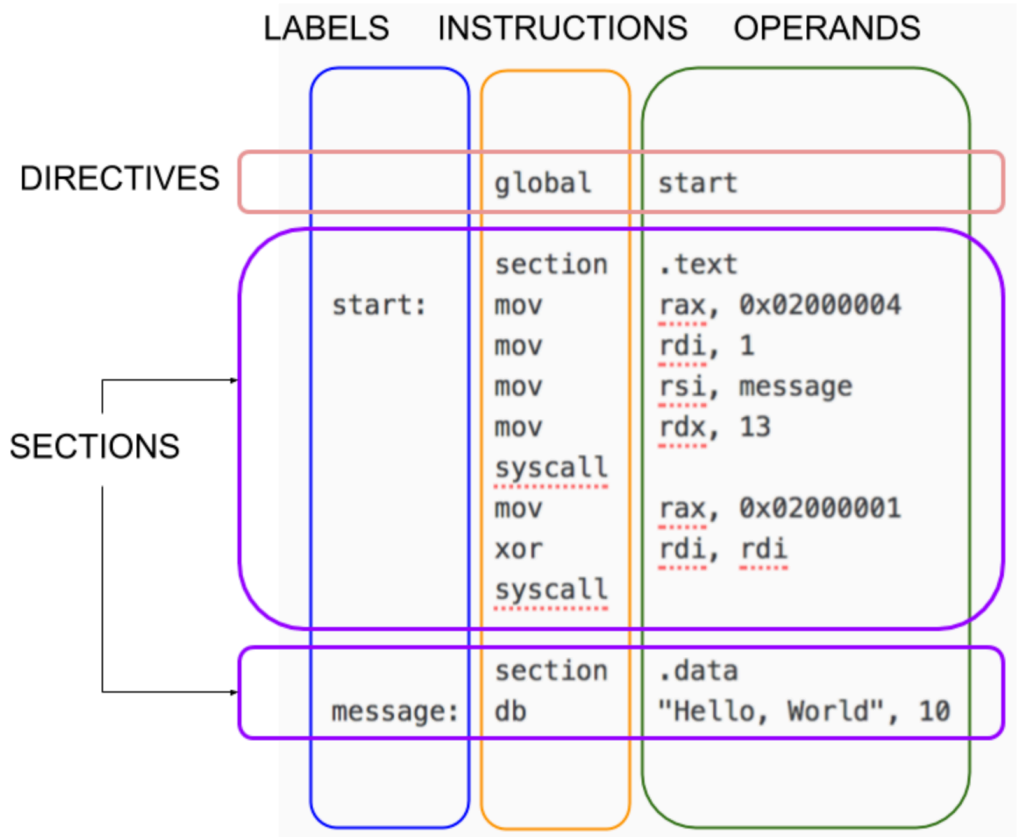
```
        section   .data
message:  db        "Hello, World", 10      ; note the newline at the end

```

hello.asm

```
; ------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit macOS only.
; To assemble and run:
;
;     nasm -fmacho64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------------

          global    start

          section   .text
start:    mov       rax, 0x02000004         ; system call for write
          mov       rdi, 1                  ; file handle 1 is stdout
          mov       rsi, message            ; address of string to output
          mov       rdx, 13                 ; number of bytes
          syscall                           ; invoke operating system to do the write
          mov       rax, 0x02000001         ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating system to exit

          section   .data
message:  db        "Hello, World", 10      ; note the newline at the end
```

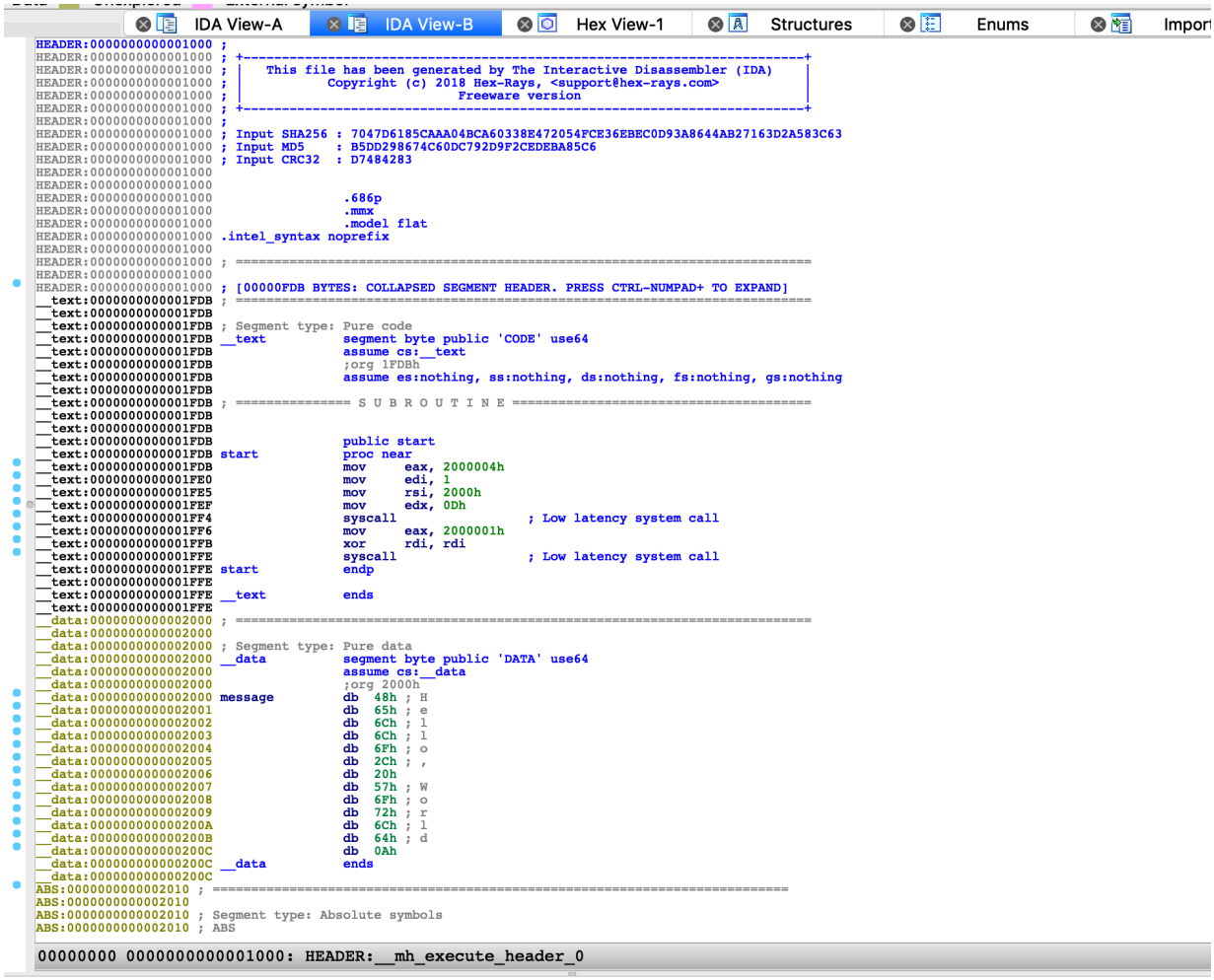
```
$ nasm -fmacho64 hello.asm && ld hello.o && ./a.out
Hello, World
```

NASM is line-based. Most programs consist of directives followed by one or more sections. Lines can have an optional label. Most lines have an instruction followed by zero or more operands.



Generally, you put code in a section called `.text` and your constant data in a section called `.data`.

```
; Segment type: Pure code
__text segment byte public 'CODE' use64
assume cs:__text
;org 1FDBh
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing



public start
start proc near
mov     eax, 2000004h
mov     edi, 1
mov     rsi, 2000h
mov     edx, 0Dh
syscall                 ; Low latency system call
mov     eax, 2000001h
xor     rdi, rdi
syscall                 ; Low latency system call
start endp

__text ends
```



https://cseweb.ucsd.edu/classes/sp11/cse141/pdf/02/S01_x86_64.key.pdf

Registers

16bit	32bit	64bit	Description	Notes
AX	EAX	RAX	The accumulator register	These can be used more or less interchangeably
BX	EBX	RBX	The base register	
CX	ECX	RCX	The counter	
DX	EDX	RDX	The data register	
SP	ESP	RSP	Stack pointer	
BP	EBP	RBP	Points to the base of the stack frame	
	Rn	RnD	(n = 8...15) General purpose registers	
SI	ESI	RSI	Source index for string operations	
DI	EDI	RDI	Destination index for string operations	
IP	EIP	RIP	Instruction Pointer	
FLAGS			Condition codes	

Different names (e.g. ax vs. eax vs. rax) refer to different parts of the same register

3

The Three Kinds of Operands

Register Operands

In this tutorial we only care about the integer registers and the xmm registers. You should already know what the registers are, but here is a quick review. The 16 integer registers are 64 bits wide and are called:

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
RAX RCX RDX RBX RSP RBP RSI RDI

(Note that 8 of the registers have alternate names.) You can treat the lowest 32-bits of each register as a register itself but using these names:

R0D R1D R2D R3D R4D R5D R6D R7D R8D R9D R10D R11D R12D R13D R14D R15D
EAX ECX EDX EBX ESP EBP ESI EDI

You can treat the lowest 16-bits of each register as a register itself but using these names:

R0W R1W R2W R3W R4W R5W R6W R7W R8W R9W R10W R11W R12W R13W R14W R15W
AX CX DX BX SP BP SI DI

You can treat the lowest 8-bits of each register as a register itself but using these names:

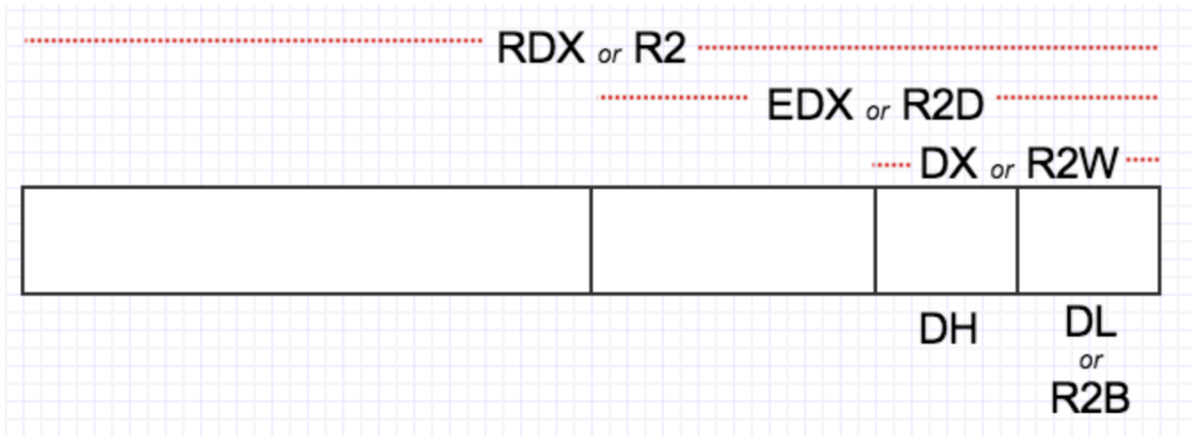
R0B R1B R2B R3B R4B R5B R6B R7B R8B R9B R10B R11B R12B R13B R14B R15B
AL CL DL BL SPL BPL SIL DIL

For historical reasons, bits 15 through 8 of R0..R3 are named:

AH CH DH BH

And finally, there are 16 XMM registers, each 128 bits wide, named:

XMM0 ... XMM15



https://cs.lmu.edu/~ray/notes/nasmtutorial/

Memory Operands

These are the basic forms of addressing:

- `[ number ]`
- `[ reg ]`
- `[ reg + reg*scale ]` *scale is 1, 2, 4, or 8 only*
- `[ reg + number ]`
- `[ reg + reg*scale + number ]`

The number is called the **displacement**; the plain register is called the **base**; the register with the scale is called the **index**.

Examples:

```
[750]                ; displacement only
[rbp]                ; base register only
[rcx + rsi*4]        ; base + index * scale
[rbp + rdx]          ; scale is 1
[rbx - 8]            ; displacement is -8
[rax + rdi*8 + 500]  ; all four components
[rbx + counter]      ; uses the address of the variable 'counter' as the displacement
```

Immediate Operands

These can be written in many ways. Here are some examples from the official docs.

```
200          ; decimal
0200         ; still decimal - the leading 0 does not make it octal
0200d        ; explicitly decimal - d suffix
0d200        ; also decimal - 0d prefex
0c8h         ; hex - h suffix, but leading 0 is required because c8h looks like a var
0xc8         ; hex - the classic 0x prefix
0hc8         ; hex - for some reason NASM likes 0h
310q         ; octal - q suffix
0q310        ; octal - 0q prefix
11001000b    ; binary - b suffix
0b1100_1000  ; binary - 0b prefix, and by the way, underscores are allowed
```

Instructions with two memory operands are extremely rare

In fact, we'll not see any such instruction in this tutorial. Most of the basic instructions have only the following forms:

`add` *reg, reg*

`add` *reg, mem*

`add` *reg, imm*

`add` *mem, reg*

`add` *mem, imm*

Defining Data and Reserving Space

These examples come from Chapter 3 of the docs. To place data in memory:

```
      db    0x55                ; just the byte 0x55
      db    0x55,0x56,0x57      ; three bytes in succession
      db    'a',0x55            ; character constants are OK
      db    'hello',13,10,'$'   ; so are string constants
      dw    0x1234              ; 0x34 0x12
      dw    'a'                 ; 0x61 0x00 (it's just a number)
      dw    'ab'                ; 0x61 0x62 (character constant)
      dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
      dd    0x12345678          ; 0x78 0x56 0x34 0x12
      dd    1.234567e20         ; floating-point constant
      dq    0x123456789abcdef0  ; eight byte constant
      dq    1.234567e20         ; double-precision float
      dt    1.234567e20         ; extended-precision float
```

There are other forms; check the NASM docs. Later.

To reserve space (without initializing), you can use the following pseudo instructions. They should go in a section called `.bss` (you'll get an error if you try to use them in a `.text` section):

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray:      resq    10              ; array of ten reals
```

Source: https://cs.lmu.edu/~ray/notes/nasmtutorial/

triangle.asm

```
; ------------------------------------------------------------------------------
; This is an OSX console program that writes a little triangle of asterisks to standard
; output. Runs on macOS only.
;
;     nasm -fmacho64 triangle.asm && gcc hola.o && ./a.out
; ------------------------------------------------------------------------------

          global    start
          section   .text
start:
          mov       rdx, output             ; rdx holds address of next byte to write
          mov       r8, 1                   ; initial line length
          mov       r9, 0                   ; number of stars written on line so far
line:
          mov       byte [rdx], '*'         ; write single star
          inc       rdx                     ; advance pointer to next cell to write
          inc       r9                      ; "count" number so far on line
          cmp       r9, r8                  ; did we reach the number of stars for this line?
          jne       line                    ; not yet, keep writing on this line
lineDone:
          mov       byte [rdx], 10          ; write a new line char
          inc       rdx                     ; and move pointer to where next char goes
          inc       r8                      ; next line will be one char longer
          mov       r9, 0                   ; reset count of stars written on this line
          cmp       r8, maxlines            ; wait, did we already finish the last line?
          jng       line                    ; if not, begin writing this line
done:
          mov       rax, 0x02000004         ; system call for write
          mov       rdi, 1                  ; file handle 1 is stdout
          mov       rsi, output             ; address of string to output
          mov       rdx, dataSize           ; number of bytes
          syscall                           ; invoke operating system to do the write
          mov       rax, 0x02000001         ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating system to exit

          section   .bss
maxlines  equ       8
dataSize  equ       44
output:   resb      dataSize
```

nasm -fmacho64 triangle.asm && ld -macosx_version_min 10.7.0 -lSystem -o triangle triangle.o && ./triangle

$ nasm -fmacho64 triangle.asm && ld -macosx_version_min 10.7.0 -lSystem -o triangle triangle.o && ./triangle

```
*
**
***
****
*****
******
```

****
****

https://stackoverflow.com/questions/44860003/how-much-bytes-does-resb-resw-resd-resq-allocates-in-nasm

RESB

https://www.nasm.us/xdoc/2.11.06/html/nasmdoc3.html#section-3.2.2

> 3.2.2 RESB and Friends: Declaring Uninitialized Data
>
> RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ are designed to be used in the BSS section of a module: they declare uninitialized storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. As stated in section 2.2.7, NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing DW ? or similar things: this is what it does instead. The operand to a RESB-type pseudo-instruction is a critical expression: see section 3.8.
>
> For example:
>
> buffer: resb 64 ; reserve 64 bytes
>
> wordvar: resw 1 ; reserve a word
>
> realarray resq 10 ; array of ten reals
>
> ymmval: resy 1 ; one YMM register
>
> zmmvals: resz 32 ; 32 ZMM registers

EQU

3.2.4 EQU: Defining Constants

EQU defines a symbol to a given constant value: when EQU is used, the source line must contain a label. The action of EQU is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. So, for example,

```
message         db      'hello, world'
msglen          equ     $-message
```

defines msglen to be the constant 12. msglen may not then be redefined later. This is not a preprocessor definition either: the value of msglen is evaluated *once*, using the value of $ (see section 3.5 for an explanation of $) at the point of definition, rather than being evaluated wherever it is referenced and using the value of $ at the point of reference.

https://www.nasm.us/xdoc/2.11.06/html/nasmdoc3.html#section-3.2.2

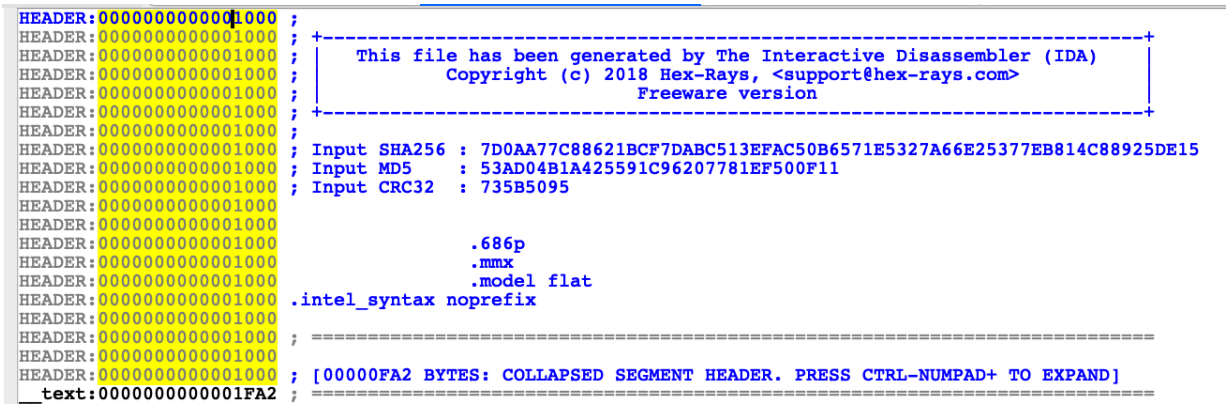
```
HEADER:0000000000001000 ;
HEADER:0000000000001000 ;
+-------------------------------------------------------------------------+
HEADER:0000000000001000 ; |   This file has been generated by The Interactive
Disassembler (IDA)    |
HEADER:0000000000001000 ; |         Copyright (c) 2018 Hex-Rays,
<support@hex-rays.com>         |
HEADER:0000000000001000 ; |                       Freeware version
|
HEADER:0000000000001000 ;
+-------------------------------------------------------------------------+
HEADER:0000000000001000 ;
HEADER:0000000000001000 ; Input SHA256 :
7D0AA77C88621BCF7DABC513EFAC50B6571E5327A66E25377EB814C88925DE1
5
HEADER:0000000000001000 ; Input MD5    :
53AD04B1A425591C96207781EF500F11
HEADER:0000000000001000 ; Input CRC32  : 735B5095
HEADER:0000000000001000
HEADER:0000000000001000
HEADER:0000000000001000              .686p
HEADER:0000000000001000              .mmx
HEADER:0000000000001000              .model flat
HEADER:0000000000001000 .intel_syntax noprefix
HEADER:0000000000001000
HEADER:0000000000001000 ;
===============================================================
===============
HEADER:0000000000001000
HEADER:0000000000001000 ; [00000FA2 BYTES: COLLAPSED SEGMENT
HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
__text:0000000000001FA2 ;
===============================================================
===============
__text:0000000000001FA2
__text:0000000000001FA2 ; Segment type: Pure code
__text:0000000000001FA2 __text         segment byte public 'CODE' use64
__text:0000000000001FA2                assume cs:__text
__text:0000000000001FA2                ;org 1FA2h
```

```
__text:0000000000001FA2                 assume es:nothing, ss:nothing, ds:nothing,
fs:nothing, gs:nothing
__text:0000000000001FA2
__text:0000000000001FA2 ; =============== S U B R O U T I N E
=======================================
__text:0000000000001FA2
__text:0000000000001FA2
__text:0000000000001FA2                 public start
__text:0000000000001FA2 start           proc near
__text:0000000000001FA2                 mov     rdx, 2000h
__text:0000000000001FAC                 mov     r8d, 1
__text:0000000000001FB2                 mov     r9d, 0
__text:0000000000001FB2 start           endp
__text:0000000000001FB2
__text:0000000000001FB8
__text:0000000000001FB8 ; =============== S U B R O U T I N E
=======================================
__text:0000000000001FB8
__text:0000000000001FB8
__text:0000000000001FB8 sub_1FB8        proc near               ; CODE XREF:
sub_1FB8+C↓j
__text:0000000000001FB8                                         ; sub_1FC6+13↓j
__text:0000000000001FB8                 mov     byte ptr [rdx], 78h
__text:0000000000001FBB                 inc     rdx
__text:0000000000001FBE                 inc     r9
__text:0000000000001FC1                 cmp     r9, r8
__text:0000000000001FC4                 jnz     short sub_1FB8
__text:0000000000001FC4 sub_1FB8        endp
__text:0000000000001FC4
__text:0000000000001FC6
__text:0000000000001FC6 ; =============== S U B R O U T I N E
=======================================
__text:0000000000001FC6
__text:0000000000001FC6
__text:0000000000001FC6 sub_1FC6        proc near
__text:0000000000001FC6                 mov     byte ptr [rdx], 0Ah
__text:0000000000001FC9                 inc     rdx
__text:0000000000001FCC                 inc     r8
__text:0000000000001FCF                 mov     r9d, 0
__text:0000000000001FD5                 cmp     r8, 8
__text:0000000000001FD9                 jle     short sub_1FB8
__text:0000000000001FD9 sub_1FC6        endp
__text:0000000000001FD9
```

```
__text:0000000000001FDB
__text:0000000000001FDB ; =============== S U B R O U T I N E
=======================================
__text:0000000000001FDB
__text:0000000000001FDB
__text:0000000000001FDB done          proc near
__text:0000000000001FDB               mov     eax, 2000004h
__text:0000000000001FE0               mov     edi, 1
__text:0000000000001FE5               mov     rsi, 2000h
__text:0000000000001FEF               mov     edx, 2Ch
__text:0000000000001FF4               syscall                 ; Low latency system call
__text:0000000000001FF6               mov     eax, 2000001h
__text:0000000000001FFB               xor     rdi, rdi
__text:0000000000001FFE               syscall                 ; Low latency system call
__text:0000000000001FFE done          endp
__text:0000000000001FFE
__text:0000000000001FFE __text        ends
__text:0000000000001FFE
__bss:0000000000002000 ;
============================================================
===============
__bss:0000000000002000
__bss:0000000000002000 ; Segment type: Uninitialized
__bss:0000000000002000 __bss          segment byte public 'BSS' use64
__bss:0000000000002000                assume cs:__bss
__bss:0000000000002000                ;org 2000h
__bss:0000000000002000                assume es:nothing, ss:nothing, ds:nothing,
fs:nothing, gs:nothing
__bss:0000000000002000 output         db    ? ;
__bss:0000000000002001                db    ? ;
__bss:0000000000002002                db    ? ;
__bss:0000000000002003                db    ? ;
__bss:0000000000002004                db    ? ;
__bss:0000000000002005                db    ? ;
__bss:0000000000002006                db    ? ;
__bss:0000000000002007                db    ? ;
__bss:0000000000002008                db    ? ;
__bss:0000000000002009                db    ? ;
__bss:000000000000200A                db    ? ;
__bss:000000000000200B                db    ? ;
__bss:000000000000200C                db    ? ;
__bss:000000000000200D                db    ? ;
__bss:000000000000200E                db    ? ;
```

```
__bss:000000000000200F                db    ? ;
__bss:0000000000002010                db    ? ;
__bss:0000000000002011                db    ? ;
__bss:0000000000002012                db    ? ;
__bss:0000000000002013                db    ? ;
__bss:0000000000002014                db    ? ;
__bss:0000000000002015                db    ? ;
__bss:0000000000002016                db    ? ;
__bss:0000000000002017                db    ? ;
__bss:0000000000002018                db    ? ;
__bss:0000000000002019                db    ? ;
__bss:000000000000201A                db    ? ;
__bss:000000000000201B                db    ? ;
__bss:000000000000201C                db    ? ;
__bss:000000000000201D                db    ? ;
__bss:000000000000201E                db    ? ;
__bss:000000000000201F                db    ? ;
__bss:0000000000002020                db    ? ;
__bss:0000000000002021                db    ? ;
__bss:0000000000002022                db    ? ;
__bss:0000000000002023                db    ? ;
__bss:0000000000002024                db    ? ;
__bss:0000000000002025                db    ? ;
__bss:0000000000002026                db    ? ;
__bss:0000000000002027                db    ? ;
__bss:0000000000002028                db    ? ;
__bss:0000000000002029                db    ? ;
__bss:000000000000202A                db    ? ;
__bss:000000000000202B                db    ? ;
__bss:000000000000202B __bss          ends
__bss:000000000000202B
ABS:0000000000002030 ;
================================================================
===============
ABS:0000000000002030
ABS:0000000000002030 ; Segment type: Absolute symbols
ABS:0000000000002030 ; ABS
ABS:0000000000002030 ; const struct mach_header_64 _mh_execute_header
ABS:0000000000002030 __mh_execute_header= 1000h
ABS:0000000000002038 dataSize         = 2Ch
ABS:0000000000002040 maxlines         = 8
ABS:0000000000002040
UNDEF:0000000000002050 ;
```

```
UNDEF:0000000000002050 ; Imports from /usr/lib/libSystem.B.dylib
UNDEF:0000000000002050 ;
UNDEF:0000000000002050 ;
================================================================
===============
UNDEF:0000000000002050
UNDEF:0000000000002050 ; Segment type: Externs
UNDEF:0000000000002050 ; UNDEF
UNDEF:0000000000002050                 extrn dyld_stub_binder:qword
UNDEF:0000000000002050
UNDEF:0000000000002050
UNDEF:0000000000002050                 end start
```

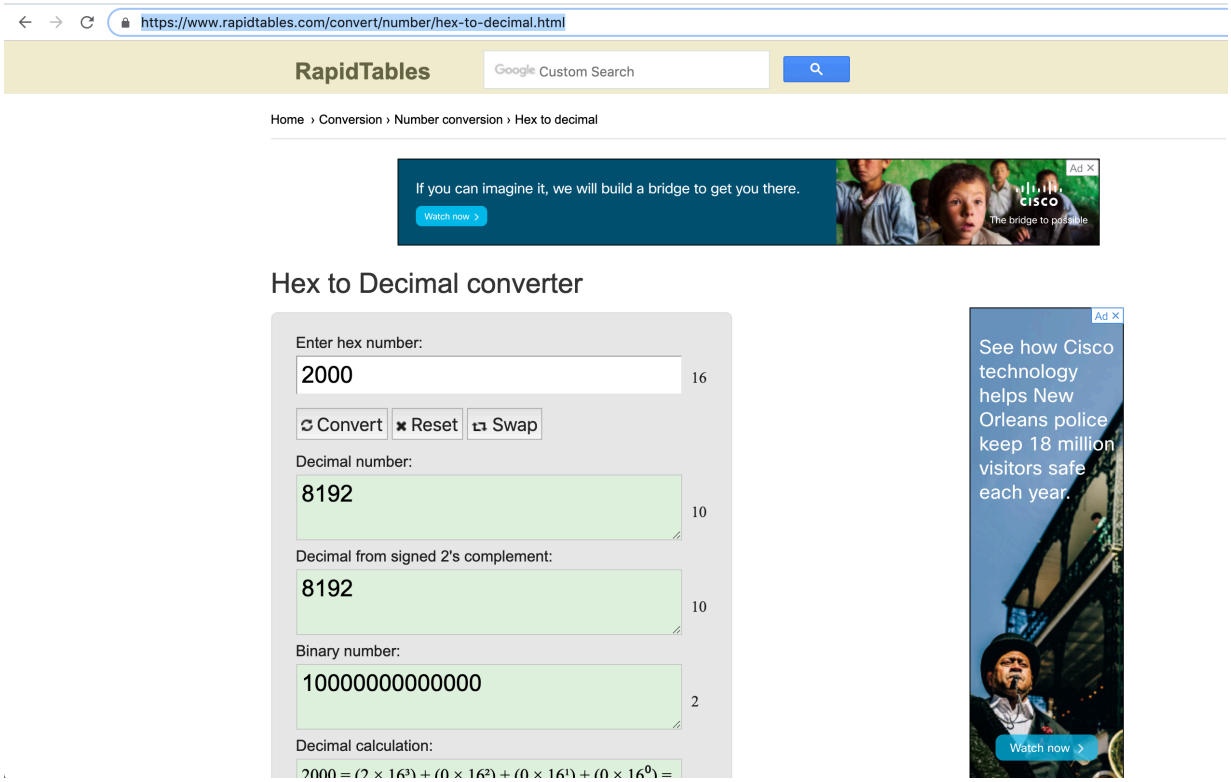


```
__text:0000000000001FA2 ; ===========================================================================
__text:0000000000001FA2
__text:0000000000001FA2 ; Segment type: Pure code
__text:0000000000001FA2 __text          segment byte public 'CODE' use64
__text:0000000000001FA2                 assume cs:__text
__text:0000000000001FA2                 ;org 1FA2h
__text:0000000000001FA2                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__text:0000000000001FA2
__text:0000000000001FA2 ; =============== S U B R O U T I N E =======================================
__text:0000000000001FA2
__text:0000000000001FA2
__text:0000000000001FA2                 public start
__text:0000000000001FA2 start           proc near
__text:0000000000001FA2                 mov     rdx, 2000h
__text:0000000000001FAC                 mov     r8d, 1
__text:0000000000001FB2                 mov     r9d, 0
__text:0000000000001FB2 start           endp
__text:0000000000001FB2
__text:0000000000001FB8
__text:0000000000001FB8 ; =============== S U B R O U T I N E =======================================
__text:0000000000001FB8
__text:0000000000001FB8
__text:0000000000001FB8 sub_1FB8        proc near               ; CODE XREF: sub_1FB8+C↓j
__text:0000000000001FB8                                         ; sub_1FC6+13↓j
__text:0000000000001FB8                 mov     byte ptr [rdx], 78h
__text:0000000000001FBB                 inc     rdx
__text:0000000000001FBE                 inc     r9
__text:0000000000001FC1                 cmp     r9, r8
__text:0000000000001FC4                 jnz     short sub_1FB8
__text:0000000000001FC4 sub_1FB8        endp
__text:0000000000001FC4
__text:0000000000001FC6
__text:0000000000001FC6 ; =============== S U B R O U T I N E =======================================
__text:0000000000001FC6
__text:0000000000001FC6
__text:0000000000001FC6 sub_1FC6        proc near
__text:0000000000001FC6                 mov     byte ptr [rdx], 0Ah
__text:0000000000001FC9                 inc     rdx
__text:0000000000001FCC                 inc     r8
__text:0000000000001FCF                 mov     r9d, 0
__text:0000000000001FD5                 cmp     r8, 8
__text:0000000000001FD9                 jle     short sub_1FB8
__text:0000000000001FD9 sub_1FC6        endp
__text:0000000000001FD9
__text:0000000000001FDB
__text:0000000000001FDB ; =============== S U B R O U T I N E =======================================
__text:0000000000001FDB
__text:0000000000001FDB
__text:0000000000001FDB done            proc near
__text:0000000000001FDB                 mov     eax, 2000004h
__text:0000000000001FE0                 mov     edi, 1
__text:0000000000001FE5                 mov     rsi, 2000h
__text:0000000000001FEF                 mov     edx, 2Ch
__text:0000000000001FF4                 syscall                 ; Low latency system call
__text:0000000000001FF6                 mov     eax, 2000001h
__text:0000000000001FFB                 xor     rdi, rdi
__text:0000000000001FFE                 syscall                 ; Low latency system call
__text:0000000000001FFE done            endp
__text:0000000000001FFE
__text:0000000000001FFE __text          ends
__text:0000000000001FFE
__bss:0000000000002000 ; ===========================================================================
__bss:0000000000002000
__bss:0000000000002000 ; Segment type: Uninitialized
__bss:0000000000002000 __bss            segment byte public 'BSS' use64
__bss:0000000000002000                  assume cs:__bss
__bss:0000000000002000                  ;org 2000h
__bss:0000000000002000                  assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__bss:0000000000002000 output           db    ? ;
__bss:0000000000002001                  db    ? ;
```

```
__bss:0000000000002000                 ;org 2000h
__bss:0000000000002000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__bss:0000000000002000 output          db    ? ;
__bss:0000000000002001                 db    ? ;
__bss:0000000000002002                 db    ? ;
__bss:0000000000002003                 db    ? ;
__bss:0000000000002004                 db    ? ;
__bss:0000000000002005                 db    ? ;
__bss:0000000000002006                 db    ? ;
__bss:0000000000002007                 db    ? ;
__bss:0000000000002008                 db    ? ;
__bss:0000000000002009                 db    ? ;
__bss:000000000000200A                 db    ? ;
__bss:000000000000200B                 db    ? ;
__bss:000000000000200C                 db    ? ;
__bss:000000000000200D                 db    ? ;
__bss:000000000000200E                 db    ? ;
__bss:000000000000200F                 db    ? ;
__bss:0000000000002010                 db    ? ;
__bss:0000000000002011                 db    ? ;
__bss:0000000000002012                 db    ? ;
__bss:0000000000002013                 db    ? ;
__bss:0000000000002014                 db    ? ;
__bss:0000000000002015                 db    ? ;
__bss:0000000000002016                 db    ? ;
__bss:0000000000002017                 db    ? ;
__bss:0000000000002018                 db    ? ;
__bss:0000000000002019                 db    ? ;
__bss:000000000000201A                 db    ? ;
__bss:000000000000201B                 db    ? ;
__bss:000000000000201C                 db    ? ;
__bss:000000000000201D                 db    ? ;
__bss:000000000000201E                 db    ? ;
__bss:000000000000201F                 db    ? ;
__bss:0000000000002020                 db    ? ;
__bss:0000000000002021                 db    ? ;
__bss:0000000000002022                 db    ? ;
__bss:0000000000002023                 db    ? ;
__bss:0000000000002024                 db    ? ;
__bss:0000000000002025                 db    ? ;
__bss:0000000000002026                 db    ? ;
__bss:0000000000002027                 db    ? ;
__bss:0000000000002028                 db    ? ;
__bss:0000000000002029                 db    ? ;
__bss:000000000000202A                 db    ? ;
__bss:000000000000202B                 db    ? ;
__bss:000000000000202B __bss           ends
__bss:000000000000202B
ABS:0000000000002030 ; ===========================================================================
ABS:0000000000002030
ABS:0000000000002030 ; Segment type: Absolute symbols
ABS:0000000000002030 ; ABS
ABS:0000000000002030 ; const struct mach_header_64 _mh_execute_header
ABS:0000000000002030 __mh_execute_header= 1000h
ABS:0000000000002038 dataSize          = 2Ch
ABS:0000000000002040 maxlines          = 8
ABS:0000000000002040
UNDEF:0000000000002050 ;
UNDEF:0000000000002050 ; Imports from /usr/lib/libSystem.B.dylib
UNDEF:0000000000002050 ;
UNDEF:0000000000002050 ; ===========================================================================
UNDEF:0000000000002050
UNDEF:0000000000002050 ; Segment type: Externs
UNDEF:0000000000002050 ; UNDEF
UNDEF:0000000000002050                 extrn dyld_stub_binder:qword
UNDEF:0000000000002050
UNDEF:0000000000002050
UNDEF:0000000000002050                 end start
```

```
UNKNOWN 0000000000002000: __bss:output
```

https://www.rapidtables.com/convert/number/hex-to-decimal.html



https://cseweb.ucsd.edu/classes/sp11/cse141/pdf/02/S01_x86_64.key.pdf

hola.asm

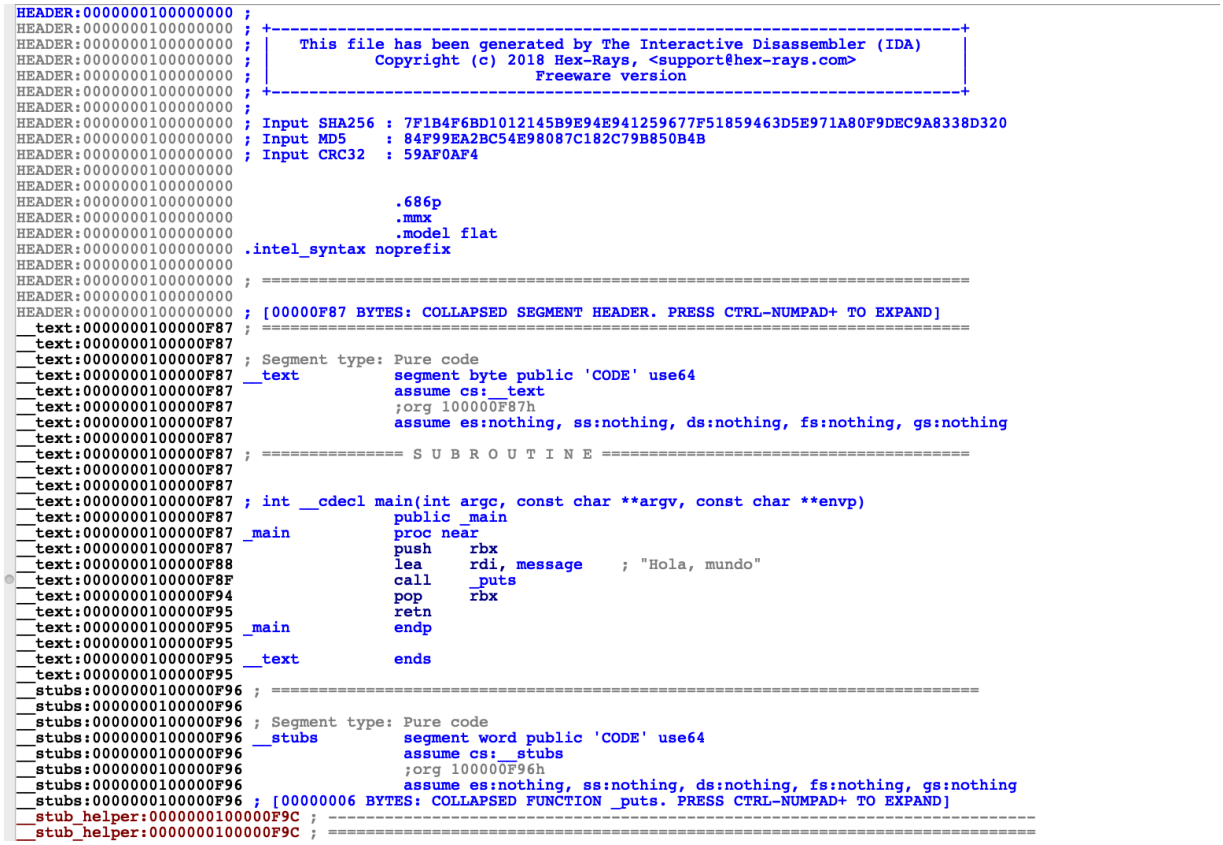
```
; ------------------------------------------------------------------------------------------
; This is an macOS console program that writes "Hola, mundo" on one line and then exits.
; It uses puts from the C library.  To assemble and run:
;
;     nasm -fmacho64 hola.asm && gcc hola.o && ./a.out
; ------------------------------------------------------------------------------------------

          global    _main
          extern    _puts

          section   .text
_main:    push      rbx                     ; Call stack must be aligned
          lea       rdi, [rel message]      ; First argument is address of message
          call      _puts                   ; puts(message)
          pop       rbx                     ; Fix up stack before returning
          ret

          section   .data
message:  db        "Hola, mundo", 0        ; C strings need a zero byte at the end
```

```
$ nasm -fmacho64 hola.asm && gcc hola.o && ./a.out
Hola, mundo
```

https://cs.lmu.edu/~ray/notes/nasmtutorial/

nasm -fmacho64 hola.asm && gcc hola.o && ./a.out

$ nasm -fmacho64 hola.asm && gcc hola.o && ./a.out
Hola, mundo



```
HEADER:0000000100000000 ;
HEADER:0000000100000000 ; +-------------------------------------------------------------------------+
HEADER:0000000100000000 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
HEADER:0000000100000000 ; |           Copyright (c) 2018 Hex-Rays, <support@hex-rays.com>           |
HEADER:0000000100000000 ; |                         Freeware version                                |
HEADER:0000000100000000 ; +-------------------------------------------------------------------------+
HEADER:0000000100000000 ;
HEADER:0000000100000000 ; Input SHA256 : 7F1B4F6BD1012145B9E94E941259677F51859463D5E971A80F9DEC9A8338D320
HEADER:0000000100000000 ; Input MD5    : 84F99EA2BC54E98087C182C79B850B4B
HEADER:0000000100000000 ; Input CRC32  : 59AF0AF4
HEADER:0000000100000000
HEADER:0000000100000000
HEADER:0000000100000000                 .686p
HEADER:0000000100000000                 .mmx
HEADER:0000000100000000                 .model flat
HEADER:0000000100000000 .intel_syntax noprefix
HEADER:0000000100000000
HEADER:0000000100000000 ; ===========================================================================
HEADER:0000000100000000
HEADER:0000000100000000 ; [00000F87 BYTES: COLLAPSED SEGMENT HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
__text:0000000100000F87 ; ===========================================================================
__text:0000000100000F87
__text:0000000100000F87 ; Segment type: Pure code
__text:0000000100000F87 __text          segment byte public 'CODE' use64
__text:0000000100000F87                 assume cs:__text
__text:0000000100000F87                 ;org 100000F87h
__text:0000000100000F87                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__text:0000000100000F87
__text:0000000100000F87 ; =============== S U B R O U T I N E =======================================
__text:0000000100000F87
__text:0000000100000F87
__text:0000000100000F87 ; int __cdecl main(int argc, const char **argv, const char **envp)
__text:0000000100000F87                 public _main
__text:0000000100000F87 _main           proc near
__text:0000000100000F87                 push    rbx
__text:0000000100000F88                 lea     rdi, message    ; "Hola, mundo"
__text:0000000100000F8F                 call    _puts
__text:0000000100000F94                 pop     rbx
__text:0000000100000F95                 retn
__text:0000000100000F95 _main           endp
__text:0000000100000F95
__text:0000000100000F95 __text          ends
__text:0000000100000F95
__stubs:0000000100000F96 ; ===========================================================================
__stubs:0000000100000F96
__stubs:0000000100000F96 ; Segment type: Pure code
__stubs:0000000100000F96 __stubs         segment word public 'CODE' use64
__stubs:0000000100000F96                 assume cs:__stubs
__stubs:0000000100000F96                 ;org 100000F96h
__stubs:0000000100000F96                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
__stubs:0000000100000F96 ; [00000006 BYTES: COLLAPSED FUNCTION _puts. PRESS CTRL-NUMPAD+ TO EXPAND]
__stub_helper:0000000100000F9C ; ---------------------------------------------------------------------------
__stub_helper:0000000100000F9C ; ===========================================================================
```

https://cs.lmu.edu/~ray/notes/nasmtutorial/

nasm -fmacho64 fib.asm && gcc fib.o && ./a.out

$ cat fib.asm

```
; ------------------------------------------------------------------------------
; A 64-bit Linux application that writes the first 90 Fibonacci numbers. To
; assemble and run:
;
;     nasm -felf64 fib.asm && gcc fib.o && ./a.out
; ------------------------------------------------------------------------------

        global  _main
        extern  _printf

        section .text
_main:
        push    rbx             ; we have to save this since we use it

        mov     ecx, 90         ; ecx will countdown to 0
        xor     rax, rax        ; rax will hold the current number
```

```
        xor     rbx, rbx              ; rbx will hold the next number
        inc     rbx                   ; rbx is originally 1
_print:
        ; We need to call printf, but we are using rax, rbx, and rcx.  printf
        ; may destroy rax and rcx so we will save these before the call and
        ; restore them afterwards.

        push    rax                   ; caller-save register
        push    rcx                   ; caller-save register

        mov     rdi, format           ; set 1st parameter (format)
        mov     rsi, rax              ; set 2nd parameter (current_number)
        xor     rax, rax              ; because printf is varargs

        ; Stack is already aligned because we pushed three 8 byte registers
        call    _printf               ; printf(format, current_number)

        pop     rcx                   ; restore caller-save register
        pop     rax                   ; restore caller-save register

        mov     rdx, rax              ; save the current number
        mov     rax, rbx              ; next number is now current
        add     rbx, rdx              ; get the new next number
        dec     ecx                   ; count down
        jnz     _print                ; if not done counting, do some more

        pop     rbx                   ; restore rbx before returning
        ret
        section   .data
format:
        db  "%20ld", 10, 0
```

fib.asm

```
; ------------------------------------------------------------------------
; A 64-bit Linux application that writes the first 90 Fibonacci numbers. To
; assemble and run:
;
;     nasm -felf64 fib.asm && gcc fib.o && ./a.out
; ------------------------------------------------------------------------

        global  main
        extern  printf

        section .text
main:
        push    rbx                     ; we have to save this since we use it

        mov     ecx, 90                 ; ecx will countdown to 0
        xor     rax, rax                ; rax will hold the current number
        xor     rbx, rbx                ; rbx will hold the next number
        inc     rbx                     ; rbx is originally 1
print:
        ; We need to call printf, but we are using rax, rbx, and rcx.  printf
        ; may destroy rax and rcx so we will save these before the call and
        ; restore them afterwards.

        push    rax                     ; caller-save register
        push    rcx                     ; caller-save register

        mov     rdi, format             ; set 1st parameter (format)
        mov     rsi, rax                ; set 2nd parameter (current_number)
        xor     rax, rax                ; because printf is varargs

        ; Stack is already aligned because we pushed three 8 byte registers
        call    printf                  ; printf(format, current_number)

        pop     rcx                     ; restore caller-save register
        pop     rax                     ; restore caller-save register

        mov     rdx, rax                ; save the current number
        mov     rax, rbx                ; next number is now current
        add     rbx, rdx                ; get the new next number
        dec     ecx                     ; count down
        jnz     print                   ; if not done counting, do some more

        pop     rbx                     ; restore rbx before returning
        ret
format:
        db  "%20ld", 10, 0
```

https://cs.lmu.edu/~ray/notes/nasmtutorial/