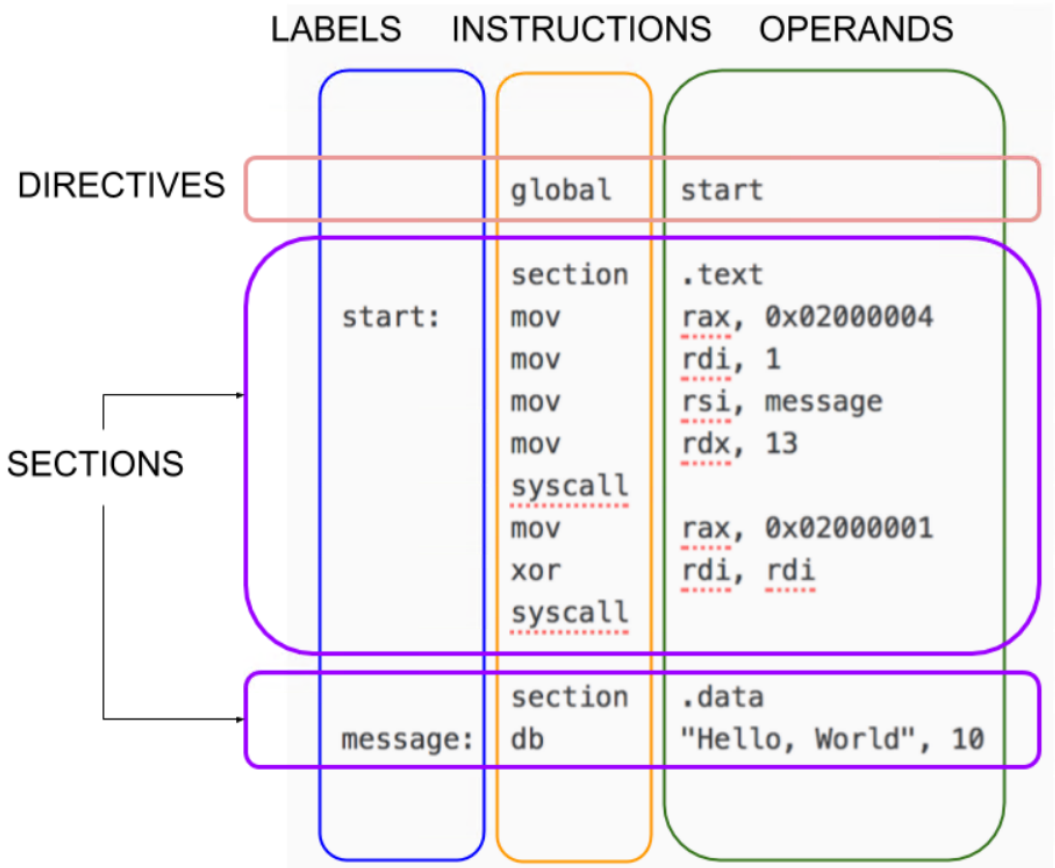

```
; ------------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------------------------

          global    _start

          section   .text
_start:   mov       rax, 1                  ; 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, 60                 ; 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 -felf64 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`.

There are hundreds of instructions. You can't learn them all at once. Just start with these:

`mov` *x*, *y*	*x* ← *y*
`and` *x*, *y*	*x* ← *x* and *y*
`or` *x*, *y*	*x* ← *x* or *y*
`xor` *x*, *y*	*x* ← *x* xor *y*
`add` *x*, *y*	*x* ← *x* + *y*
`sub` *x*, *y*	*x* ← *x* − *y*
`inc` *x*	*x* ← *x* + 1
`dec` *x*	*x* ← *x* − 1
`syscall`	Invoke an operating system routine
`db`	A pseudo-instruction that declares bytes that will be in memory when the program runs

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
```

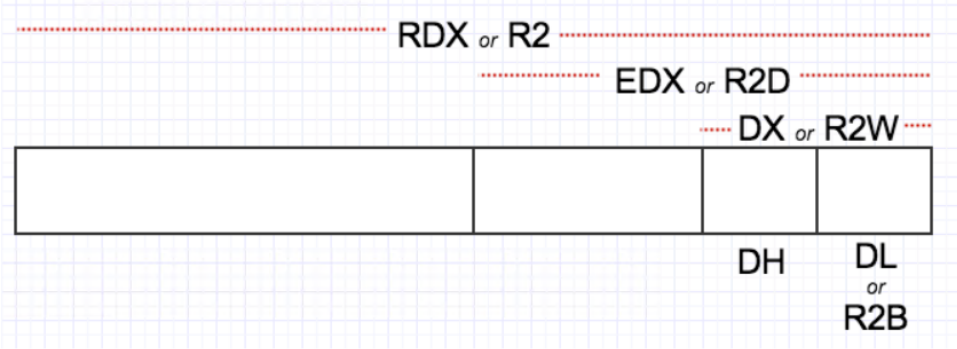
For historical reasons, bits 15 through 8 of `R0`..`R3` are named:

```
AH  CH  DH  BH
```

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

```
XMM0 ... XMM15
```

Study this picture; hopefully it helps:



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 ]`

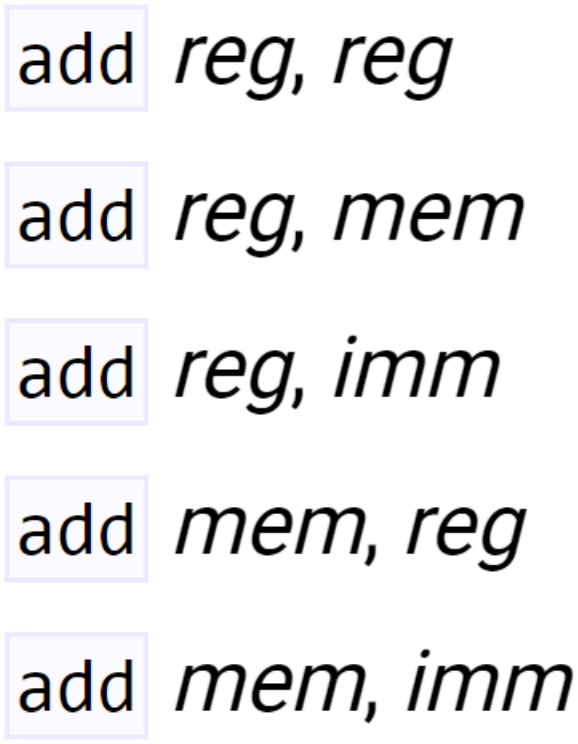
The number is called the **displacement**; the plain register is called the **base**; the register with the scale is called the **index**.

```
[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
```



```
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
```

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
```

```
          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 triangle.o && ./a.out
*
**
***
****
*****
******
*******
********
```

- `cmp` does a comparison
- `je` jumps to a label if the previous comparison was equal. We also have `jne` (jump if not equal), `jl` (jump if less), `jnl` (jump if not less), `jg` (jump if greater), `jng` (jump if not greater), `jle` (jump if less or equal), `jnle` (jump if not less or equal), `jge` (jump if greater or equal), `jnge` (jump if not greater or equal), and many more.
- `equ` is actually not a real instruction. It simply defines an abbreviation for the assembler itself to use. (This is a profound idea.)
- The `.bss` section is for *writable* data.

Writing standalone programs with just system calls is cool, but rare. We would like to use the good stuff in the C library.

Remember how in C execution "starts" at the function `main`? That's because the C library actually has the `_start` label inside itself! The code at `_start` does some initialization, then it calls `main`, then it does some clean up, then it issues the system call for exit. So you just have to implement `main`. We can do that in assembly!

```
          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
```