FREE eBook

LEARNING Assembly Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#assembly

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: assembly-language

It is an unofficial and free Assembly Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Assembly Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Assembly Language

Remarks

Assembly is a general name used for many human-readable forms of machine code. It naturally differs a lot between different CPUs (Central Processing Unit), but also on single CPU there may exist several incompatible dialects of Assembly, each compiled by different assembler, into the identical machine code defined by the CPU creator.

If you want to ask question about your own Assembly problem, always state what HW and which assembler you are using, otherwise it will be difficult to answer your question in detail.

Learning Assembly of single particular CPU will help to learn basics on different CPU, but every HW architecture can have considerable differences in details, so learning ASM for new platform can be close to learning it from scratch.

Links:

X86 Assembly Wikibook

Examples

Introduction

Assembly language is a human readable form of machine language or machine code which is the actual sequence of bits and bytes on which the processor logic operates. It is generally easier for humans to read and program in mnemonics than binary, octal or hex, so humans typically write code in assembly language and then use one or more programs to convert it into the machine language format understood by the processor.

EXAMPLE:

```
mov eax, 4
cmp eax, 5
je point
```

An assembler is a program that reads the assembly language program, parses it, and produces the corresponding machine language. It is important to understand that unlike a language like C++ that is a single language defined in standard document, there are many different assembly languages. Each processor architecture, ARM, MIPS, x86, etc has a different machine code and thus a different assembly language. Additionally, there are sometimes multiple different assembly languages for the same processor architecture. In particular, the x86 processor family has two popular formats which are often referred to as **gas syntax** (`gas` is the name of the executable for the GNU Assembler) and **Intel syntax** (named after the originator of the x86 processor family).

They are different but equivalent in that one can typically write any given program in either syntax.

Generally, the inventor of the processor documents the processor and its machine code and creates an assembly language. It's common for that particular assembly language to be the only one used, but unlike compiler writers attempting to conform to a language standard, the assembly language defined by the inventor of the processor is usually but not always the version used by the people who write assemblers.

There are two general types of processors:

- CISC (Complex Instruction Set Computer): have many different and often complex machine language instructions
- RISC (Reduced Instruction set Computers): by contrast, has fewer and simpler instructions

For an assembly language programmer, the difference is that a CISC processor may have a great many instructions to learn but there are often instructions suited for a particular task, while RISC processors have fewer and simpler instructions but any given operation may require the assembly language programmer to write more instructions to get the same thing done.

Other programming languages compilers sometimes produce assembler first, which is then compiled into machine code by calling an assembler. For example, *gcc* using its own *gas* assembler in final stage of compilation. Produced machine code is often stored in *object* files, which can be linked into executable by the linker program.

A complete "toolchain" often consists of a compiler, assembler and linker. One can then use that assembler and linker directly to write programs in assembly language. In the GNU world the *binutils* package contains the assembler and linker and related tools; those who are solely interested in assembly language programming do not need *gcc* or other compiler packages.

Small microcontrollers are often programmed purely in assembly language or in a combination of assembly language and one or more higher level languages such as C or C++. This is done because one can often use the particular aspects of the **instruction set architecture** for such devices to write more compact, efficient code than would be possible in a higher level language and such devices often have limited memory and registers. Many microprocessors are used in **embedded systems** which are devices other than general purpose computers that happen to have a microprocessor inside. Examples of such embedded systems are televisions, microwave ovens and the engine control unit of a modern automobile. Many such devices have no keyboard or screen, so a programmer generally writes the program on a general purpose computer, runs a **cross-assembler** (so called because this kind of assembler produces code for a different kind of processor than the one on which it runs) and/or a **cross-compiler** and **cross linker** to produce machine code.

There are many vendors for such tools, which are as varied as the processors for which they produce code. Many, but not all processors also have an open source solution like GNU, sdcc, llvm or other.

Machine code

Machine code is term for the data in particular native machine format, which are directly processed by the machine - usually by the processor called *CPU* (Central Processing Unit).

Common computer architecture (*von Neumann architecture*) consist of general purpose processor (CPU), general purpose memory - storing both program (ROM/RAM) and processed data and input and output devices (I/O devices).

The major advantage of this architecture is relative simplicity and universality of each of components - when compared to computer machines before (with hard-wired program in the machine construction), or competing architectures (for example the *Harvard architecture* separating memory of program from memory of data). Disadvantage is a bit worse general performance. Over long run the universality allowed for flexible usage, which usually outweighed the performance cost.

How does this relate to machine code?

Program and data are stored in these computers as numbers, in the memory. There's no genuine way to tell apart code from data, so the operating systems and machine operators give the CPU hints, at which entry point of memory starts the program, after loading all the numbers into memory. The CPU then reads the instruction (number) stored at entry point, and processing it rigorously, sequentially reading next numbers as further instructions, unless the program itself tells CPU to continue with execution elsewhere.

For example a two 8 bit numbers (8 bits grouped together are equal to 1 byte, that's an unsigned integer number within 0-255 range): `60 201`, when executed as code on Zilog Z80 CPU will be processed as two instructions: `INC a` (incrementing value in register `a` by one) and `RET` (returning from sub-routine, pointing CPU to execute instructions from different part of memory).

To define this program a human can enter those numbers by some memory/file editor, for example in hex-editor as two bytes: `3C C9` (decimal numbers 60 and 201 written in base 16 encoding). That would be *programming in **machine code***.

To make the task of CPU programming easier for humans, an **Assembler** programs were created, capable to read text file containing something like:

```
subroutineIncrementA:
    INC   a
    RET

dataValueDefinedInAssemblerSource:
    DB    60          ; define byte with value 60 right after the ret
```

outputting byte hex-numbers sequence `3C C9 3C`, wrapped around with optional additional numbers specific for target platform: marking which part of such binary is executable code, where is the entry point for program (the first instruction of it), which parts are encoded data (not executable), etc.

Notice how the programmer specified the last byte with value 60 as "data", but from CPU perspective it does not differ in any way from `INC a` byte. It's up to the executing program to

correctly navigate CPU over bytes prepared as instructions, and process data bytes only as data for instructions.

Such output is usually stored in a file on storage device, loaded later by OS (*Operating System - a machine code already running on the computer, helping to manipulate with the computer*) into memory ahead of executing it, and finally pointing the CPU on the entry point of program.

The CPU can process and execute only machine code - but any memory content, even random one, can be processed as such, although result may be random, ranging from "*crash*" detected and handled by OS up to accidental wipe of data from I/O devices, or damage of sensitive equipment connected to the computer (not a common case for home computers :)).

The similar process is followed by many other high level programming languages, compiling the **source** (human readable text form of program) into numbers, either representing the machine code (native instructions of CPU), or in case of interpreted/hybrid languages into some general language-specific virtual machine code, which is further decoded into native machine code during execution by interpreter or virtual machine.

Some compilers use the Assembler as intermediate stage of compilation, translating the source firstly into Assembler form, then running assembler tool to get final **machine code** out of it (GCC example: run `gcc -S helloworld.c` to get an assembler version of C program `helloworld.c`).

Hello world for Linux x86_64 (Intel 64 bit)

```
section .data
    msg db "Hello world!",10      ; 10 is the ASCII code for a new line (LF)

section .text
    global _start

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, 13
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

If you want to execute this program, you first need the Netwide Assembler, `nasm`, because this code uses its syntax. Then use the following commands (assuming the code is in the file `helloworld.asm`). They are needed for assembling, linking and executing, respectively.

- `nasm -felf64 helloworld.asm`
- `ld helloworld.o -o helloworld`
- `./helloworld`

The code makes use of Linux's `sys_write` syscall. Here you can see a list of all syscalls for the x86_64 architecture. When you also take the man pages of write and exit into account, you can

translate the above program into a C one which does the same and is much more readable:

```
#include <unistd.h>

#define STDOUT 1

int main()
{
    write(STDOUT, "Hello world!\n", 13);
    _exit(0);
}
```

Just two commands are needed here for compilation and linking (first one) and executing:

- `gcc helloworld_c.c -o helloworld_c.`
- `./helloworld_c`

Hello World for OS X (x86_64, Intel syntax gas)

```
.intel_syntax noprefix

.data

.align 16
hello_msg:
    .asciz "Hello, World!"

.text

.global _main
_main:
    push rbp
    mov rbp, rsp

    lea rdi, [rip+hello_msg]
    call _puts

    xor rax, rax
    leave
    ret
```

Assemble:

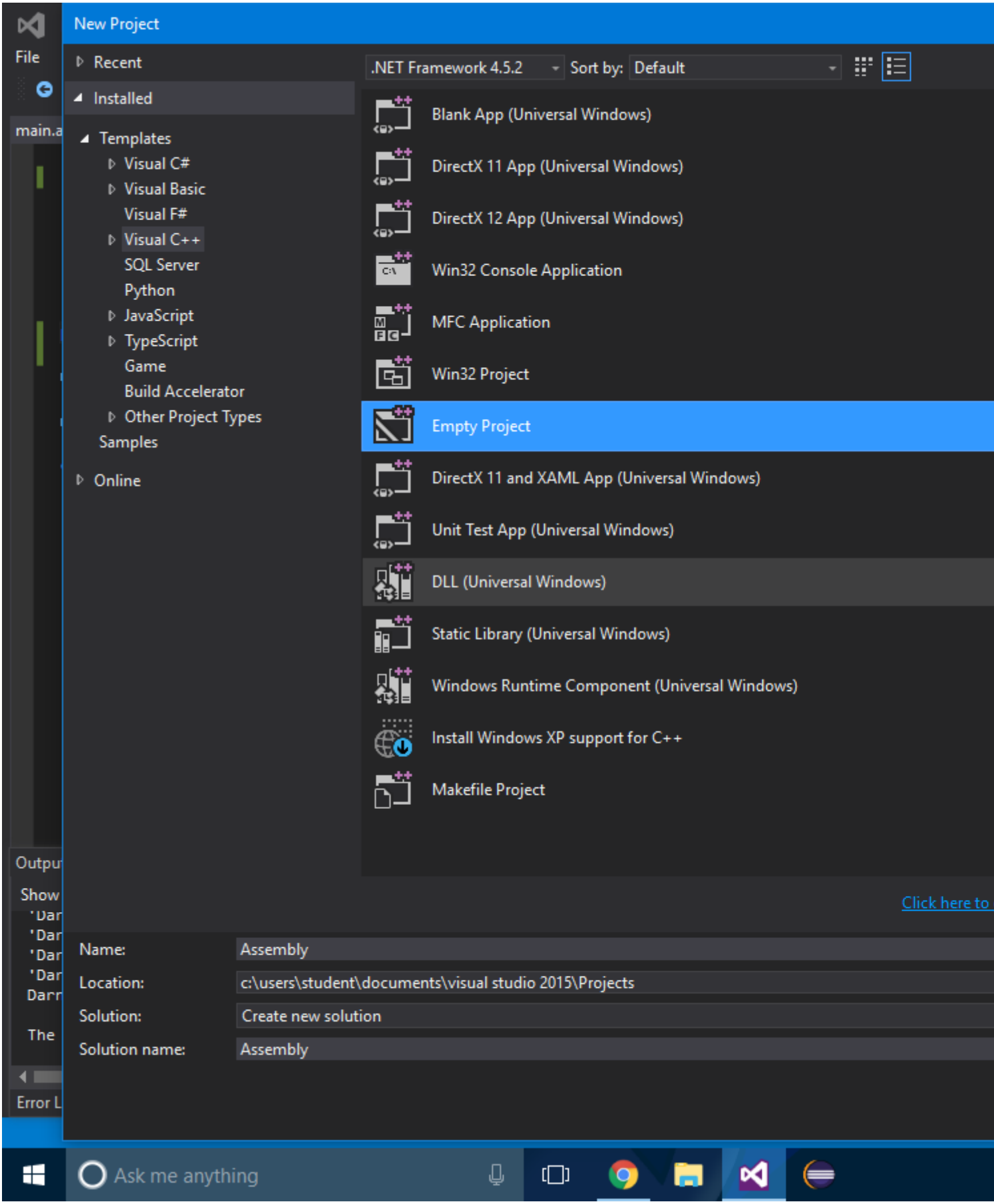
```
clang main.s -o hello
./hello
```

Notes:

- The use of system calls is discouraged as the system call API in OS X is not considered stable. Instead, use the C library. (Reference to a Stack Overflow question)
- Intel recommends that structures larger than a word begin on a 16-byte boundary. (Reference to Intel documentation)
- The order data is passed into functions through the registers is: rdi, rsi, rdx, rcx, r8, and r9. (Reference to System V ABI)

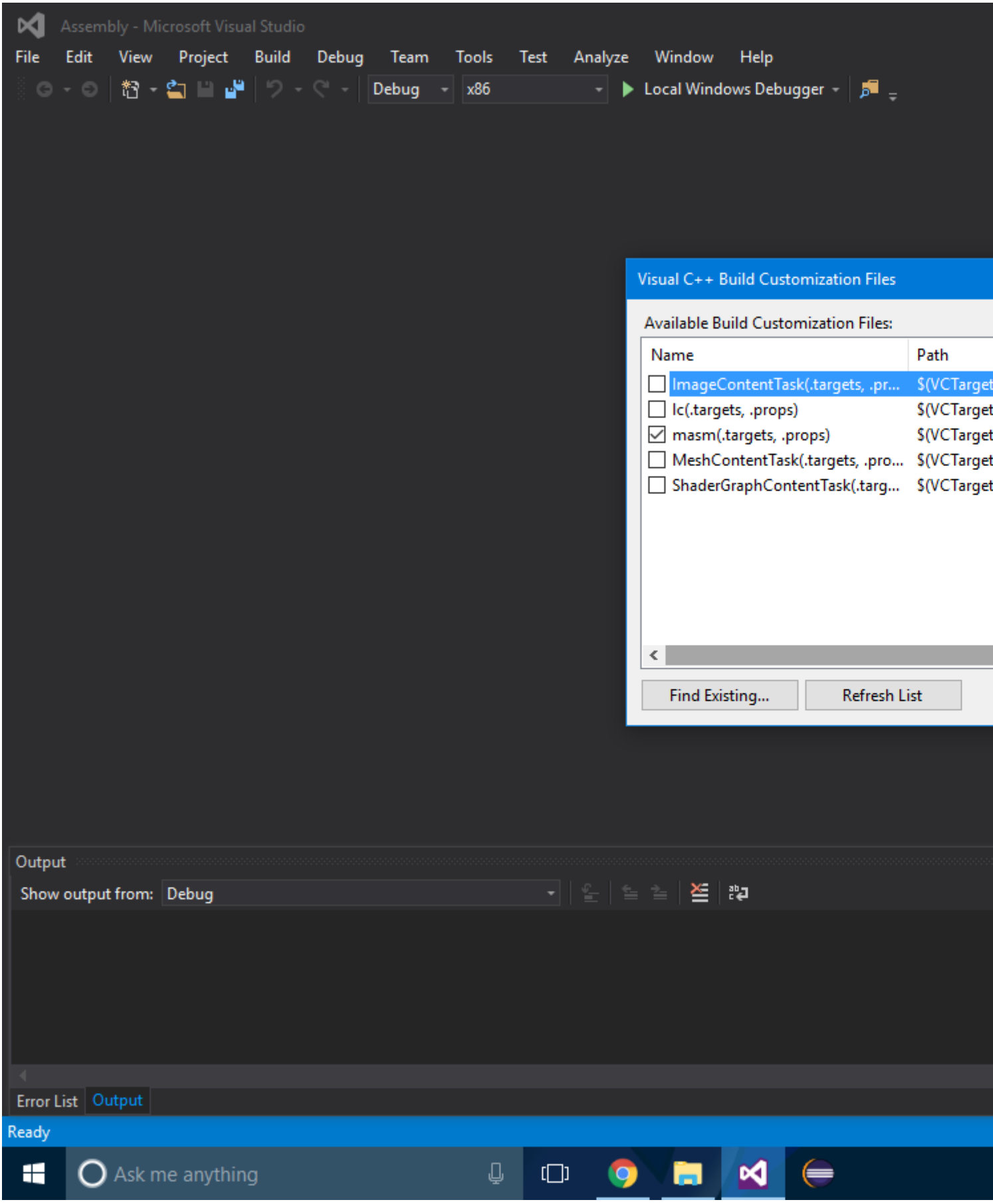
Executing x86 assembly in Visual Studio 2015

Step 1: Create an empty project via **File -> New Project**.



Step 2: Right click the project solution and select **Build Dependencies->Build Customizations**.

Step 3: Check the checkbox *".masm"*.



Step 4: Press the button *"ok"*.

Step 5: Create your assembly file and type in this:

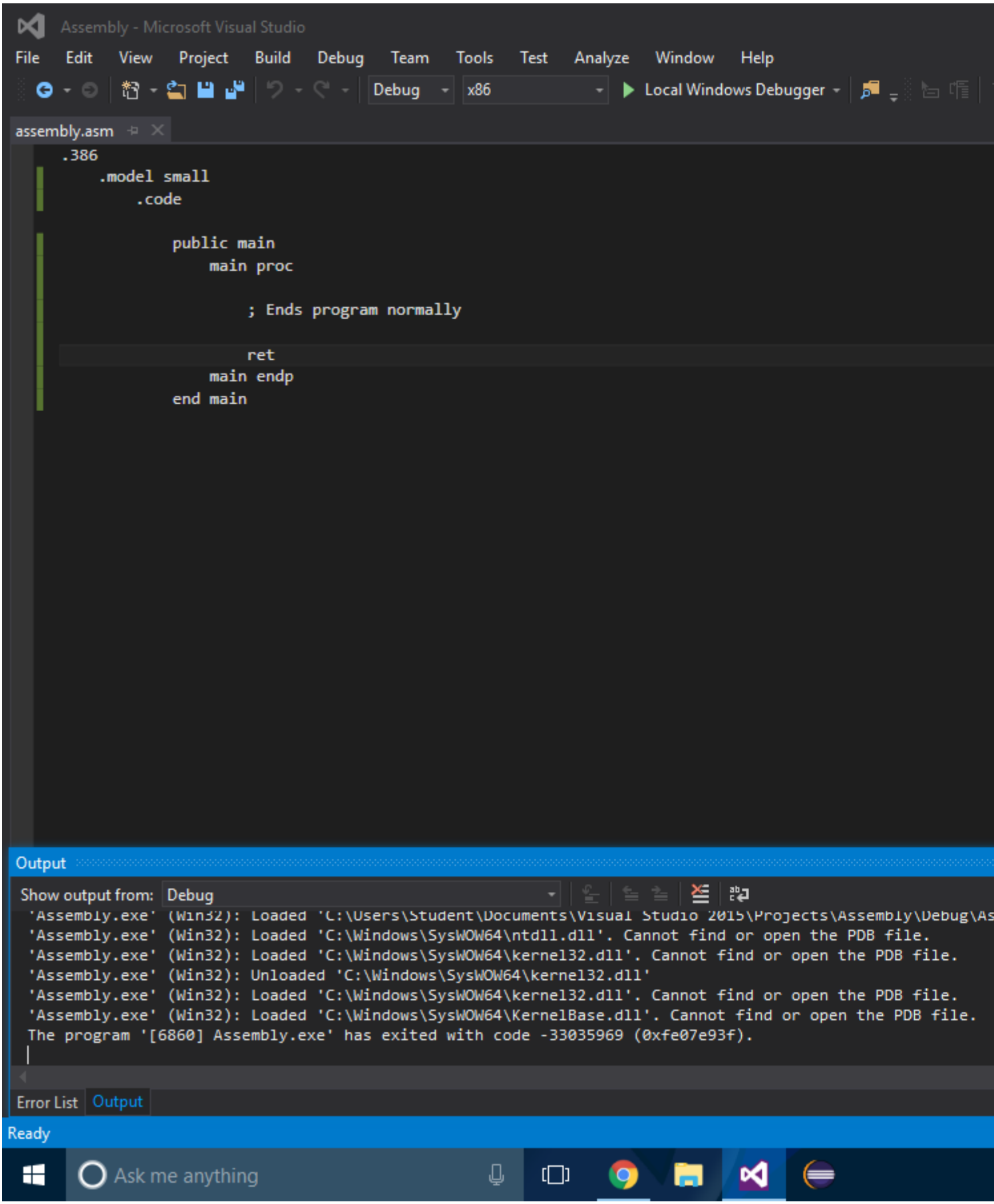
```
.386
    .model small
        .code

            public main
                main proc

                    ; Ends program normally

                    ret
                main endp
            end main
```

Step 6: Compile!



Read Getting started with Assembly Language online:
https://riptutorial.com/assembly/topic/1358/getting-started-with-assembly-language

Chapter 2: Flow Control

Introduction

Every piece of non-trivial software needs flow-control structures to divert program flow according to conditions. Assembly being the lowest-level programming language provides only *primitives* for control structures. Typically, machine operations affect *flags* in the CPU, and *conditional branches/jumps* implement the flow control. In assembly, all higher-level control structures need to be constructed from such primitives.

Examples

Trivial IF-THEN-ELSE in m68k Assembly

```
; IF d0 == 10 GO TO ten, ELSE GO TO other
    CMP    #10,d0        ; compare register contents to immediate value 10
                         ; instruction affects the zero flag
    BEQ    ten           ; branch if zero flag set
other:
    ; do whatever needs to be done for d0 != 10
    BRA    afterother    ; unconditionally jump across IF case
ten:
    ; do whatever needs to be done for d0 == 10
afterother:
    ; continue normal common program flow
```

Which instructions are affecting which flags, and which conditional branches (that might also be based on specific *combinations of flags*) are available, depends very much on your chosen CPU and should be looked up in the manuals.

FOR ... NEXT in Z80 Assembly

The Z80 has a specific instruction to implement loop counts: `DJNZ`standing for "decrement B register and jump if not zero". So, B is the register of choice to implement loops on this processor. FOR...NEXT needs to be implemented "backwards", because the register counts down to zero. Other CPUs (like the 8086, this CPU uses the CX register as loop counter) might have similar specific loop counter registers and instructions, some other CPUs allow loop commands with arbitrary registers (m68k has a DBRA instruction that works with any data register).

```
; Trivial multiplication (by repeated adding, ignores zero in factors, so
; not recommended for general use)
;
; inputs:    A = Factor 1
;            B = Factor 2
;
; output:    A = Factor 1 * Factor 2
;
; Pseudo code
; C = A : A = 0 : FOR B = Factor 2 DOWNTO 0 : A = A + C : NEXT B
```

```
mul:
     LD    C,A        ; Save Factor 1 in C register
     XOR   A          ; Clear accumulator
mLoop:
     ADD   A,C        ; Add Factor 1 to accumulator
     DJNZ  mLoop      ; Do this Factor 2 times
     RET              ; return to caller
```

If-statement in Intel-syntax assembly

```
section .data
    msg_eq db 'Equal', 10
    len_eq equ $ - msg_eq

    msg_le db 'Less than', 10
    len_le equ $ - msg_le

    msg_gr db 'Greater than', 10
    len_gr equ $ - msg_gr ; Length of msg_gr
section .text
    global _main ; Make the _main label global for linker
_main:
    cmp 4, 5 ; Compare 4 and 5
    je _equal ; je = jump if equal
    jl _less ; jl = jump if less
    jg _greater ; jg = jump if greater
exit:
    ret ; Return
_equal:
    ; Whatever code here
    mov rax, 0x2000004 ; sys_write, 4 for linux
    mov rdi, 1 ; STDOUT
    mov rsi, msg_eq
    mov rdi, len_eq

    syscall

    jmp exit ; Exit
_less:
    ; Whatever code here
    mov rax, 0x2000004
    mov rdi, 1
    mov rsi, msg_le
    mov rdi, len_le

    syscall

    jmp exit
_greater:
    ; Whatever code here

    mov rax, 0x2000004
    mov rdi, 1
    mov rsi, msg_gr
    mov rdi, len_gr

    syscall
    jmp exit
```

Loop while condition is true in Intel syntax assembly

```
section .data
    msg db 'Hello, world!', 0xA
    len equ $ - msg
section .text
global _main
_main:
    mov rax, 0 ; This will be the current number
    mov rcx, 10 ; This will be the last number

_loop:
    cmp rax, rcx
    jl .loopbody ; Jump to .loopbody if rax < rcx
    jge _exit ; Jump to _exit if rax ≥ rcx
.loopbody:
    push rax ; Store the rax value for later use

    mov rax, 0x2000004 ; 4 for Linux
    mov rdi, 1 ; STDOUT
    mov rsi, msg
    mov rdx, len

    syscall

    pop rax ; Take it back to rax

    inc rax ; Add 1 to rax. This is required since the loop must have an ending.

    jmp _loop ; Back to loop
_exit:
    ret    ; Return
```

This will execute `.loopbody` as long as `rax < rcx`.

Read Flow Control online: https://riptutorial.com/assembly/topic/8172/flow-control

Chapter 3: Interrupts

Remarks

Why do we need Interrupts

Lets imagine: Our computer is connected to a keypad. We want to enter something. When we press the key nothing happens because the computer is dealing with different things and doesnt notice that we want something from him. We need Interrupts!

Interrupts are triggered by software (*INT* 80h) or hardware (keypress), they behave like a *Call* (they jump to a specific location, execute code and jump back again).

Examples

Working with Interrupts on the Z80:

The Z80 has no Interrupt table like modern processors. The Interrupts all execute the same code. In Interrupt Mode 1, they execute the code in a specific unchangeable location. In Interrupt Mode 2, they execute the code from the Pointer register I points to. The Z80 has got a timer, that triggers the Interrupt all ~0.007s.

```
EI      ;enables Interrupts
DI      ;disables Interrupts
IM 1    ;sets the Normal Interrupt Mode


IM 2    ;sets the Advanced Interrupt Mode
LD I,$99;sets the Interrupt Pointer to $99 (just possible in IM 2)
```

Read Interrupts online: https://riptutorial.com/assembly/topic/6555/interrupts

Chapter 4: Linux elf64 examples not using glibc

Examples

User Interface

I would venture to say that 80% of the processing that goes on in modern computing systems does not require user interaction, such as kernel code for Linux, OSX and Windows. For those that do, there are two fundamentals which are interactivity via keyboard (*pointing devices*) and console. This example and others in my series are oriented around text based console (VT100 emulation) and keyboard.

In and of itself, this example is very simple, but it is an essential building block toward more complex algorithms.

Subrtx.asm

```
        STDIN    equ        0
       STDOUT    equ        1

     SYS_READ    equ        0
    SYS_WRITE    equ        1

    global  gets, strlen, print, atoq

            section .text
```

As this is intended exclusively for keyboard, the probability of errors is next to none. I would imagine most often, program will be able to contemplate buffer size to circumvent buffer overrun, but that is not guaranteed due to indirection.

```
; ============================================================================
; Accept canonical input from operator for a maximum of EDX bytes and replace
; terminating CR with NULL.

;        ENTER: RSI = Pointer to input buffer
;               EDX = Maximum number of characters

;        LEAVE: EAX = Number of characters entered
;               R11 = Modified by syscall, all others preserved.

;        FLAGS:  ZF = Null entry, NZ otherwise.
; ----------------------------------------------------------------------------

     gets:  push    rcx
            push    rdi
```

```
            xor     eax, eax            ; RAX = SYS_READ
            mov     edi, eax            ; RDI = STDIN
            syscall

; TODO:    Should probably do some error trapping here, especially for
;            buffer overrun, but I'll see if it becomes an issue over time.

            dec     eax                 ; Bump back to CR
            mov     byte [rsi+rax], 0   ; Replace it with NULL

            pop     rdi
            pop     rcx
            ret
```

To begin with, this was intended to circumvent the need to either code or manually calculate a strings length for write(2). Then I decided to incorporate a delimiter, now it can be used to scan for any character (0 - FF). This opens the possibility for word wrapping text for example, so the label ***strlen*** is a bit of a misnomer as one would generally think the result is going to be the number of visible character.

```
; ============================================================================
; Determine length, including terminating character EOS. Result may include
; VT100 escape sequences.

;        ENTER: RDI = Pointer to ASCII string.
;               RCX   Bits 31 - 08 = Max chars to scan (1 - 1.67e7)
;                           07 - 00 = Terminating character (0 - FF)

;        LEAVE: RAX = Pointer to next string (optional).

;        FLAGS:  ZF = Terminating character found, NZ otherwise (overrun).
; ____________________________________________________________________________

   strlen:  push    rcx                 ; Preserve registers used by proc so
            push    rdi                 ; it's non-destructive except for RAX.

            mov      al, cl             ; Byte to scan for in AL.
            shr     ecx, 8              ; Shift max count into bits 23 - 00

; NOTE: Probably should check direction flag here, but I always set and
;       reset DF in the process that is using it.

            repnz   scasb               ; Scan for AL or until ECX = 0
            mov     rax, rdi            ; Return pointer to EOS + 1

            pop     rdi                 ; Original pointer for proglogue
            jz      $ + 5               ; ZF indicates EOS was found
            mov     rax, rdi            ; RAX = RDI, NULL string
            pop     rcx

            ret
```

The intent to this procedure is to simplify loop design in the calling procedure.

```
; ============================================================================
; Display an ASCIIZ string on console that may have embedded VT100 sequences.
```

```
;        ENTER: RDI = Points to string

;        LEAVE: RAX = Number of characters displayed, including EOS
;                   = Error code if SF
;               RDI = Points to byte after EOS.
;               R11 = Modified by syscall all others preserved

;        FLAGS:  ZF = Terminating NULL was not found. NZ otherwise
;                SF = RAX is negated syscall error code.
;------------------------------------------------------------------------------

    print:  push    rsi
            push    rdx
            push    rcx

            mov     ecx, -1 << 8        ; Scan for NULL
            call    strlen
            push    rax                 ; Preserve point to next string
            sub     rax, rdi            ; EAX = End pntr - Start pntr
            jz      .done

     ; size_t = write (int STDOUT, char *, size_t length)

            mov     edx, eax            ; RDX = length
            mov     rsi, rdi            ; RSI = Pointer
            mov     eax, SYS_WRITE
            mov     edi, eax            ; RDI = STDOUT
            syscall
            or      rax, rax            ; Sets SF if syscall error

; NOTE:    This procedure is intended for console, but in the event STDOUT is
;        redirected by some means, EAX may return error code from syscall.

    .done:  pop     rdi                 ; Retrieve pointer to next string.
            pop     rcx
            pop     rdx
            pop     rsi

            ret
```

Finally an example of how these functions can be used.

Generic.asm

```
global  _start

    extern  print, gets, atoq

    SYS_EXIT  equ   60
         ESC  equ   27

       BSize  equ   96

            section .rodata
   Prompt:  db  ESC, '[2J'      ; VT100 clear screen
            db  ESC, '[4;11H'   ;   "   Position cursor to line 4 column 11
            db  'ASCII -> INT64 (binary, octal, hexidecimal, decimal), '
            db  'Packed & Unpacked BCD and floating point conversions'
```

```
            db  10, 10, 0, 9, 9, 9, '=> ', 0
            db  10, 9, 'Bye'
            db  ESC, '[0m'      ; VT100 Reset console
            db  10, 10, 0

            section .text
   _start:  pop    rdi
            mov    rsi, rsp
            and    rsp, byte 0xf0       ; Align stack on 16 byte boundary.

            call   main
            mov    rdi, rax             ; Copy return code into ARG0

            mov    eax, SYS_EXIT
            syscall

; int main ( int argc, char *args[] )

     main:  enter   BSize, 0             ; Input buffer on stack
            mov     edi, Prompt
            call    print
            lea     rsi, [rbp-BSize]    ; Establish pointer to input buffer
            mov     edx, BSize          ; Max size for read(2)

    .Next:  push    rdi                 ; Save pointer to "=> "
            call    print
            call    gets
            jz      .done

            call    atoq                ; Convert string pointed to by RSI

            pop     rdi                 ; Restore pointer to prompt
            jmp     .Next

    .done:  call    print               ; RDI already points to "Bye"
            xor     eax, eax
            leave
            ret
```

Makefile

```
OBJECTS = Subrtx.o Generic.o

Generic : $(OBJECTS)
    ld -oGeneric $(OBJECTS)
    readelf -WS Generic

Generic.o : Generic.asm
     nasm -g -felf64 Generic.asm

Subrtx.o : Subrtx.asm
    nasm -g -felf64 Subrtx.asm

clean:
    rm -f $(OBJECTS) Generic
```

Read Linux elf64 examples not using glibc online: https://riptutorial.com/assembly/topic/7059/linux-

elf64-examples-not-using-glibc

Chapter 5: Registers

Remarks

What are Registers?

The processor can operate upon numeric values (numbers), but these have to be stored somewhere first. The data are stored mostly in memory, or inside the instruction opcode (which is stored usually in memory too), or in special on-chip memory placed directly in processor, which is called **register**.

To work with value in register, you don't need to address it by address, but special mnemonic "names" are used, like for example `ax` on x86, or `A` on Z80, or `r0` on ARM.

Some processors are constructed in a way, where almost all registers are equal and can be used for all purposes (often RISC group of processors), others have distinct specialization, when only some registers may be used for arithmetic (*"accumulator"* on early CPUs) and other registers for memory addressing only, etc.

This construction using memory directly on the processor chip has huge performance implication, adding two numbers from registers storing it back to register is usually done in shortest possible time by that processor (Example on ARM processor: `ADD r2,r0,r1` sets `r2` to `(r0 + r1)` value, in single processor cycle).

On the contrary, when one of the operands is referencing a memory location, the processor may stall for some time, waiting for the value to arrive from the memory chip (on x86 this can range from zero wait for values in L0 cache to hundreds of CPU cycles when the value is not in any cache and has to be read directly from memory DRAM chip).

So when programmer is creating some data processing code, she usually wants to have all data during processing in registers to get best performance. If that's not possible, and memory reads/writes are required, then those should be minimised, and form a pattern which cooperates with caches/memory architecture of the particular platform.

The native size of register in bits is often used to group processors, like *Z80* being *"8 bit processor"*, and *80386* being *"32 bit processor"* - although that grouping is rarely a clear cut. For example *Z80* operates also with pairs of registers, forming native 16 bit value, and 32 bit *80686* CPU has MMX instructions to work with 64 bit registers natively.

Examples

Zilog Z80 registers

Registers: 8 bit: `A`, `B`, `C`, `D`, `E`, `H`, `L`, `F`, `I`, `R`, 16 bit: `SP`, `PC`, `IX`, `IY`, and shadows of some 8b registers: `A'`, `B'`, `C'`, `D'`, `E'`, `H'`, `L'` and `F'`.

Most of the 8 bit registers can be used also in pairs as 16 bit registers: `AF`, `BC`, `DE` and `HL`.

`SP` is *stack pointer*, marking the bottom of stack memory (used by `PUSH`/`POP`/`CALL`/`RET` instructions).
`PC` is *program counter*, pointing to the currently executed instruction.
`I` is *Interrupt register*, supplying high byte of vector table address for `IM 2` interrupt mode.
`R` is *refresh register*, it increments each time the CPU fetches an opcode (or opcode prefix).
Some unofficial instructions exist on some Z80 processors to manipulate 8bit parts of `IX` as `IXH:IXL` and `IY` as `IYH:IYL`.

Shadow variants can't be directly accessed by any instruction, the `EX AF,AF'` instruction will swap between `AF` and `AF'`, and `EXX` instruction will swap `BC,DE,HL` with `BC',DE',HL'`.

Loading value into a register:

```
; from other register
LD   I,A        ; copies value in A into I (8 bit)
LD   BC,HL      ; copies value in HL into BC (16 bit)
; directly with value encoded in instruction machine code
LD   B,d8       ; 8b value d8 into B
LD   DE,d16     ; 16b value d16 into DE
; from a memory (ROM/RAM)
LD   A,(HL)     ; value from memory addressed by HL into A
LD   A,(a16)    ; value from memory with address a16 into A
LD   HL,(a16)   ; 16b value from memory with address a16 into HL
POP  IX         ; 16b value popped from stack into IX
LD   A,(IY+a8)  ; IX and IY allows addressing with 8b offset
; from I/O port (for writing value at I/O port use "OUT")
IN   A,(C)      ; reads I/O port C, value goes to A
```

Correct combinations of possible source and destination operands are limited (for example `LD H,(a16)` does not exist).

Storing value into a memory:

```
LD   (HL),D     ; value D stored into memory addressed by HL
LD   (a16),A    ; value A into memory with address a16
LD   (a16),HL   ; value HL into 16b of memory with address a16
LD   (IX+a8),d8 ; value d8 into memory at address IX+a8
LD   (IY+a8),B  ; value B into memory at address IY+a8
; specials ;)
PUSH DE         ; 16b value DE pushed to stack
CALL a16        ; while primarily used for execution branching
  ; it also stores next instruction address into stack
```

x86 Registers

In the 32-bit world, the general-purpose registers fall into three general classes: the 16-bit general-purpose registers, the 32-bit extended general-purpose registers, and the 8-bit register halves. These three classes do not represent three entirely distinct sets of registers at all. The 16-bit and 8-bit registers are actually names of regions *inside* the 32-bit registers. Register growth in the x86 CPU family has come about by *extending* registers existing in older CPUs

There are eight 16-bit general-purpose registers: AX, BX, CX, DX, BP, SI, DI, and SP; and you can place any value in them that may be expressed in 16 bits or fewer.

When Intel expanded the x86 architecture to 32 bits in 1986, it doubled the size of all eight registers and gave them new names by prefixing an E in front of each register name, resulting in EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.

With x86_64 came another doubling of register size, as well as the addition of some new registers. These registers are 64 bits wide and are named (slash used to show alternate register name): RAX/r0, RBX/r3, RCX/r1, RDX/r2, RBP/r5, RSI/r6, RDI/r7, RSP/r4, R8, R9, R10, R11, R12, R13, R14, R15.

While the general purpose registers can be technically used for anything, each register also has an alternate/main purpose:

- AX (accumulator) is used in arithmetic operations.
- CX (counter) is used in the shift and rotate instructions, and used for loops.
- DX (data) is used in arithmetic and I/O operations.
- BX (base) used as a pointer to data (specifically as an offset to the DS segment register when in segmented mode).
- SP (stack) points to the top of the stack.
- BP (stack base) points to the base of the stack.
- SI (source) points to a source in memory for stream operations (e.g. `lodsb`).
- DI (destination) points to a destination in memory for stream operations (e.g. `stosb`).

Segment registers, used in segmented mode, point to different segments in memory. Each 16-bit segment register gives a view to 64k (16 bits) of data. After a segment register has been set to point to a block of memory, registers (such as `BX`, `SI`, and `DI`) can be used as offsets to the segment register so specific locations in the 64k space can be accessed.

The six segment registers and their uses are:

Register	Full name	Description
SS	Stack Segment	Points to the stack
CS	Code Segment	Used by the CPU to fetch the code
DS	Data Segment	Default register for MOV operations
ES	Extra Segment	Extra data segment
FS	Extra Segment	Extra data segment
GS	Extra Segment	Extra data segment

x64 Registers

The x64 architecture is the evolution of the older x86 architecture, it kept compatibility with its predecessor (x86 registers are still available) but it also introduced new features:

- Registers have now a capacity of 64 bits;
- There are 8 more general-purpose registers;
- Segment registers are forced to 0 in 64 bits mode;
- The lower 32, 16 and 8 bits of each register are now available.

General-purpose

Register	Name	Subregisters(bits)
RAX	Accumulator	EAX(32), AX(16), AH(8), AL(8)
RBX	Base	EBX(32), BX(16), BH(8), BL(8)
RCX	Counter	ECX(32), CX(16), CH(8), CL(8)
RDX	Data	EDX(32), DX(16), DH(8), DL(8)
RSI	Source	ESI(32), SI(16), SL(8)
RDI	Destination	EDI(32), DI(16), DL(8)
RBP	Base pointer	EBP(32), BP(16), BPL(8)
RSP	Stack pointer	ESP(32), SP(16), SPL(8)
R8-R15	New registers	R8D-R15D(32), R8W-R15W(16), R8B-R15B(8)

Note

The suffixes used to address the lower bits of the new registers stand for:

- B byte, 8 bits;
- W word, 16 bits;
- D double word, 32 bits.

Read Registers online: https://riptutorial.com/assembly/topic/4802/registers

Chapter 6: The Stack

Remarks

The stack of computers is like a stack of books. *PUSH* adds one to the top and *POP* takes the uppermost away. Like in real life the stack cannot be endless, so it has maximum size. The stack can be used for sorting algorithms, to handle a bigger amount of data or to safe values of registers while doing another operation.

Examples

Zilog Z80 Stack

The register `sp` is used as *stack pointer*, pointing to the last stored value into stack ("top" of stack). So `EX (sp),hl` will exchange value of `hl` with the value on top of stack.

Contrary to "top" word, the stack grows in memory by decreasing the `sp`, and releases ("pops") values by increasing `sp`.

For `sp` = `$4844` with values `1`, `2`, `3` stored on stack (the `3` being pushed onto stack as last value, so being at top of it), the memory will look like this:

```
|    address | value bytes | comment (btw, all numbers are in hexadecimal)
| ---------- | ----------- | ---------------------------------
|       4840 | ?? ??       | free stack spaces to be used by next push/call
|       4842 | ?? ??       | or by interrupt call! (don't expect values to stay here)
| sp -> 4844 | 03 00       | 16 bit value "3" on top of stack
|       4846 | 02 00       | 16 bit value "2"
|       4848 | 01 00       | 16 bit value "1"
|       484A | ?? ??       | Other values in stack (up to it's origin)
|       484C | ?? ??       | like for example return address for RET instruction
```

Examples, how instructions work with stack:

```
    LD   hl,$0506
    EX   (sp),hl           ; $0003 into hl, "06 05" bytes at $4844
    POP  bc                ; like: LD c,(sp); INC sp; LD b,(sp); INC sp
                           ; so bc is now $0506, and sp is $4846
    XOR  a                 ; a = 0, sets zero and parity flags
    PUSH af                ; like: DEC sp; LD (sp),a; DEC sp; LD (sp),f
                           ; so at $4844 is $0044 (44 = z+p flags), sp is $4844
    CALL $8000             ; sp is $4842, with address of next ins at top of stack
                           ; pc = $8000 (jumping to sub-routine)
                           ; after RET will return here, the sp will be $4844 again
    LD   (L1+1),sp         ; stores current sp into LD sp,nn instruction (self modification)
    DEC  sp                ; sp is $4843
L1  LD   sp,$1234          ; restores sp to $4844 ($1234 was modified)
    POP  de                ; de = $0044, sp = $4846
    POP  ix                ; ix = $0002, sp = $4848
    ...
```

```
...
ORG  $8000
RET                     ; LD pc,(sp); INC sp; INC sp
                        ; jumps to address at top of stack, "returning" to caller
```

Summary: `PUSH` will store value on top of stack, `POP` will fetch value from top of stack, it's a *LIFO* (last in, first out) queue. `CALL` is same as `JP`, but it also pushes address of next instruction after `CALL` at top of stack. `RET` is similar to `JP` also, popping the address from stack and jumping to it.

Warning: when interrupts are enabled, the `sp` must be valid during interrupt signal, with enough free space reserved for interrupt handler routine, as the interrupt signal will store the return address (actual `pc`) before calling handler routine, which may store further data on stack as well. Any value ahead of `sp` may be thus modified "unexpectedly", if interrupt happens.

Advanced trick: on Z80 with `PUSH` taking 11 clock cycles (11t) and `POP` taking 10t, the unrolled `POP`/`PUSH` trough all registers, including `EXX` for shadow variants, was the fastest way to copy block of memory, even faster than unrolled `LDI`. But you had to time the copy in between interrupt signals to avoid memory corruption. Also unrolled `PUSH` was the fastest way to fill memory with particular value on ZX Spectrum (again with the risk of corruption by Interrupt, if not timed properly, or done under `DI`).

Read The Stack online: https://riptutorial.com/assembly/topic/4957/the-stack

Credits

S. No	Chapters	Contributors
1	Getting started with Assembly Language	Community, Edward, FedeWar, godisgood4, old_timer, Ped7g, Pichi Wuana, sigalor, stackptr, TheFrenchPlays Hd Micraftn, Zopesconk
2	Flow Control	SpilledMango, tofro
3	Interrupts	Jonas W.
4	Linux elf64 examples not using glibc	Shift_Left
5	Registers	FedeWar, godisgood4, Jonas W., Ped7g, Pichi Wuana, SirPython
6	The Stack	Jonas W., Ped7g