**Сучасне використання Асемблера.**

Сьогодні асемблер використовують у чотирьох напрямках:

1. **Операційні системи та компілятори.** Сучасні операційні системи, наприклад, дистрибутиви Linux, пишуть на C/C++, але є фрагменти на асемблері. Деякі фрагменти g++ та компілятори для BASIC та Fortran теж написані машинно-орієнтованою мовою.
2. **Вбудовані системи та драйвери.** На асемблері пишуть драйвери під Windows для архітектури процесорів x86 та програми для AVR-мікроконтролерів та Arduino.
3. **Кібербезпека та хакінг.** За допомогою асемблера хакери зламують програмне забезпечення, а розробники пишуть на ньому захист від злому. Іноді трапляються і віруси на асемблері. Їх важче виявити, і вони набагато ефективніші за високорівневі.
4. **Віртуальні машини/емулятори**. Віртуальні машини тісно взаємодіють із ОС, тому частково написані на асемблері. Як, наприклад, LLVM чи Surface Duo Emulator.

Знати асемблер потрібно всім розробникам. Він допомагає налагоджувати програми, коли звичайні засоби не працюють. Розробнику простіше оптимізувати код, якщо розуміє, як процесор обробляє інструкції.

(У 2022 році асемблер увірвався до топ-10 мов програмування за версією TIOBE The TIOBE («The Importance Of Being Earnest») Programming Community index є показником популярності мов програмування. Індекс оновлюється раз на місяць. Рейтинги базуються на кількості підготовлених інженерів у всьому світі та кількості розроблених курсів. Для підрахунку рейтингів використовуються такі популярні пошукові системи, як Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube і Baidu. Важливо зауважити, що індекс TIOBE не стосується найкращої мови програмування чи мови, на якій написано більшість рядків коду. https://www.tiobe.com/tiobe-index/ )

Асемблери - це ціла група машинно-орієнтованих мов програмування. Набір команд (ISA) та архітектура конкретного асемблера залежить від типу процесора. Тому для різних процесорів команди також будуть різними — єдиного стандарту мови не існує.

Для процесора x86-x64, є більше десятка різних асемблер компіляторів. Вони відрізняються різними наборами функцій та синтаксисом та можуть компілювати програми для різних операційних систем (див. табл.1)

Таблиця 1. Порівняння компіляторів

| Назва компілятора | Windows | DOS | Linux | BSD | QNX | MacOS, работающий на процессоре Intel/AMD |
|---|---|---|---|---|---|---|
| **FASM** | x | x | x | x | | |
| **GAS** | x | x | x | x | x | x |
| **GoAsm** | x | | | | | |
| **HLA** | x | | x | | | |
| **MASM** | x | x | | | | |
| **NASM** | x | x | x | x | x | x |
| **RosAsm** | x | | | | | |
| **TASM** | x | x | | | | |

## Підтримка 16 біт

Якщо асемблер підтримує DOS, він підтримує і 16-разрядные команди. Всі асемблери дають можливість писати код, який використовує 16-розрядні операнди. 16-розрядна підтримка означає можливість створення коду, що працює у 16-розрядній сегментованій моделі пам'яті (порівняно з 32-розрядною моделлю, що використовується більшістю сучасних операційних систем).

## Підтримка 64 біт

За винятком TASM, який не підтримує в повному обсязі навіть 32-розрядні програми, всі інші діалекти підтримують розробку 64-розрядних додатків.

## Переносимість програм

Навіть на одному процесорі ви можете зіткнутися з проблемами переносимості. Наприклад, якщо ви плануєте компілювати та використовувати свої програми на асемблері під різними операційними системами. Наприклад, NASM та FASM можна використовувати у тих операційних системах, які вони підтримують.

Для різних операційних систем тексти програм також будуть мати різний вигляд:

Приклад програми "Hello, world!" для операційної системи DOS

```
section .text
org 0x100
        mov     ah, 0x9
        mov     dx, hello
        int     0x21

        mov     ax, 0x4c00
        int     0x21

section .data
hello:  db 'Hello, world!', 13, 10, '$'
```

Приклад програми "Hello, world!" для операційної системи Linux

```
global _start

section .text
_start:
        mov     eax, 4 ; write
        mov     ebx, 1 ; stdout
        mov     ecx, msg
        mov     edx, msg.len
        int     0x80   ; write(stdout, msg, strlen(msg));

        xor     eax, msg.len ; invert return value from write()
        xchg eax, ebx ; value for exit()
        mov     eax, 1 ; exit
        int     0x80   ; exit(...)

section .data
msg:    db      "Hello, world!", 10
.len:   equ     $ - msg
```

Приклад програми "Hello, world!" для операційної системи Microsoft Windows:

```
global _main
extern _MessageBoxA@16
extern _ExitProcess@4

section code use32 class=code
_main:
        push    dword 0      ; UINT uType = MB_OK
        push    dword title  ; LPCSTR lpCaption
        push    dword banner ; LPCSTR lpText
        push    dword 0      ; HWND hWnd = NULL
        call    _MessageBoxA@16

        push    dword 0      ; UINT uExitCode
```

```
        call    _ExitProcess@4

section data use32 class=data
        banner: db 'Hello, world!', 0
        title:  db 'Hello', 0
```

A 64-bit program for Apple OS X that inputs a keystroke and shows it on the screen:

```
global _start

section .data

        query_string:           db      "Enter a character:  "
        query_string_len:       equ     $ - query_string
        out_string:                     db      "You have input:  "
        out_string_len:         equ     $ - out_string

section .bss

        in_char:                        resw 4

section .text

_start:

        mov     rax, 0x2000004          ; put the write-system-call-code into
register rax
        mov     rdi, 1                          ; tell kernel to use stdout
        mov     rsi, query_string       ; rsi is where the kernel expects to find
the address of the message
        mov     rdx, query_string_len   ; and rdx is where the kernel expects to
find the length of the message
        syscall

        ; read in the character
        mov     rax, 0x2000003          ; read system call
        mov     rdi, 0                          ; stdin
        mov     rsi, in_char            ; address for storage, declared in section
.bss
        mov     rdx, 2                          ; get 2 bytes from the kernel's
buffer (one for the carriage return)
        syscall

        ; show user the output
        mov     rax, 0x2000004          ; write system call
        mov     rdi, 1                          ; stdout
        mov     rsi, out_string
        mov     rdx, out_string_len
        syscall

        mov     rax, 0x2000004          ; write system call
        mov     rdi, 1                          ; stdout
        mov     rsi, in_char
        mov     rdx, 2                          ; the second byte is to apply the
carriage return expected in the string
        syscall
```

```
        ; exit system call
        mov     rax, 0x2000001          ; exit system call
        xor     rdi, rdi
        syscall
```

### Компілятор NASM

NASM в основному виводить об'єктні файли, які, як правило, не є виконуваними самі по собі. Єдиним винятком із цього є файли .com. Для перекладу об’єктних файлів у виконувані програми необхідно використовувати відповідний компонувальник, наприклад утиліту Visual Studio "LINK" для Windows або ld для Unix-подібних систем.