1 Лабораторна робота 2. Знайомство з Асемблер

1.1 Мета роботи

- Ознайомитися з основами мови програмування Асемблер
- Створити найпростішу програму на Асемблері
- Скомпілювати і запустити найпростішу програму на Асемблері

1.2 Завдання на роботу

1.2.1 Налаштування середовища для роботи з Асемблер

Асемблер - це низькорівнева мова програмування, яка тісно пов'язана з архітектурою комп'ютера. Вона дозволяє програмісту напряму взаємодіяти з апаратним забезпеченням комп'ютера.

Для виконання роботи необхідно встановити утиліти для асемблювання та лінкування.

Асемблювання (assembly) та лінкування (linking) — це дві важливі стадії процесу компіляції програмного коду, зокрема в мовах програмування низького рівня (як-от C чи C++).

Асемблювання полягає в перетворенні коду на асемблері (assembly code), який є низькорівневою мовою програмування, в машинний код (інструкції, що виконуються процесором).

Для операційних систем Windows та Linux, типово для асемблювання використовується утиліта nasm (Netwide Assembler), в той час як для MacOS використовується утиліта as.

Лінкування відбувається після асемблювання і полягає в об'єднанні кількох окремих частин машинного коду (наприклад, окремі модулі або бібліотеки) в один виконуваний файл.

Лінкер — це програма, яка об'єднує всі необхідні модулі та бібліотеки і створює остаточний виконуваний файл.

Для Linux та MacOS використовується лінкер ld, в той час як для Windows - link.

Щоб встановити nasm для Windows, необхідно скачати інсталяційний пакет з https://www.nasm.us/pub/nasm/releasebuilds/2.16.03/win64/. Щоб використовувати лінкер, необхідно встановити Visual Studio 2022 https://visualstudio.microsoft.com/downloads/ або Windows SDK https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/.

Щоб використовувати as та ld на MacOS, достатньо встановити утиліту gcc:

```
brew install gcc
```

Щоб встановити лінкер ld та асемблер nasm для Linux, виконайте наступні команди:

```
apt install gcc nasm
```

Після встановлення необхідних утиліт, можна переходити до написання найпростішої програми.

1.2.2 Основні поняття в Асемблері

Для початку необхідно розглянути основні поняття в Асемблері, такі як регістри, інструкції, тощо.

Регістри - це швидкі області пам'яті всередині процесора. Вони використовуються для зберігання даних та адрес. Основними типами регістрів для x86_64 архітектури є:

- регістри загального призначення (RAX, RBX, RCX, RDX, тощо)
- Індексні регістри (RSI, RDI)
- Вказівники стеку (RSP)
- Лічильник команд (RIP).

Розглянемо деякі регістри детальніше.

Індексні регістри RSI (Register Source Index) та RDI (Register Destination Index) - це 64-бітні регістри, які часто використовуються для роботи з рядками та масивами. RSI зазвичай вказує на джерело даних, а RDI - на місце призначення. Наприклад, при копіюванні рядка, RSI може вказувати на початок вихідного рядка, а RDI - на початок цільового рядка.

Вказівник стеку RSP (Register Stack Pointer) - це 64-бітний регістр, який завжди вказує на верхівку стеку. Стек - це область пам'яті для тимчасового зберігання даних, передачі параметрів функцій та збереження адрес повернення. При додаванні до стеку (push), RSP зменшується; при видаленні (pop) - збільшується. Правильне управління стеком критично важливе для коректної роботи програми.

Лічильник команд RIP (Register Instruction Pointer) - це 64-бітний регістр, що вказує на адресу наступної інструкції для виконання. Лічильник команд автоматично оновлюється процесором після виконання кожної інструкції. Інструкції переходу (jump) та виклику функцій (call) напряму змінюють значення RIP.

Розуміння роботи регістрів, стеку та лічильника команд є ключовим для ефективного програмування на асемблері. Це дозволяє вам оптимізувати код,

керувати потоком виконання програми та ефективно взаємодіяти з пам'яттю на найнижчому рівні.

Інструкції - це команди, які виконує процесор. Основні приклади таких інструкцій:

- MOV - переміщення даних
- ADD - додавання
- SUB - віднімання
- CALL - виклик функції
- CMP - порівняння
- JMP/JE/JZ - перехід між інструкціями
- Логічні операції (AND, OR, XOR, TEST, тощо)

Тепер давайте розглянемо структуру програми на асемблері. Типова програма на асемблері складається з декількох секцій, які ви можете знайти на Лістингу 1.1.

Лістинг 1.1 - Структура програми на Асемблері

```
section .data
  ; оголошення змінних


section .text
  global _start          ; точка входу в програму


_start:
  ; код програми
```

Далі ми розглянемо декілька прикладів найпростіших програм для різних операційних систем і архітектур - для MacOS, Linux та Windows. Вашим завданням для розділів 1.2.3 - 1.2.5 буде уважно проаналізувати структуру коду для кожної з операційних систем та їх відмінності. У розділах 1.2.6 - 1.27 необхідно буде виконати практичне завдання.

1.2.3 Приклад програми для MacOS з ARM64 чіпом

Отже, для початку давайте розглянемо приклад програми для MacOS з ARM64 (або ще називають AArch64) чіпом, яку ви можете знайти на Лістингу 1.2. Уважно проаналізуйте структуру коду.

Лістинг 1.2 - Програма для MacOS з ARM64 чіпом

```
.global _start      // Позначає _start як глобальний символ,
                    // доступний для лінкера
.align 2            // Вирівнює наступний код за адресою,
                    // кратною 4
```

```
.text                // Початок секції коду
_start:
  // Налаштування параметрів для системного виклику write
  mov x0, #1              //Файловий дескриптор: 1 - (stdout,
                          //стандартний вивід)
  adrp x1, message@PAGE   //Завантаження старших бітів адреси
                          //повідомлення
  add x1, x1, message@PAGEOFF //Додавання зміщення для отримання
                              //повної адреси
  mov x2, #7     //Довжина повідомлення(7 символів включно з \n)

  // Виконання системного виклику write
  mov x16, #4     //Номер системного виклику для write
  svc #0x80       //Викликати системний виклик (supervisor call)

  // Вихід з програми
  mov x0, #0      // Статус виходу: 0 (успішне завершення)
  mov x16, #1     // Номер системного виклику для exit
  svc #0x80       // Викликати системний виклик exit

.data                // Початок секції даних
message:
  .ascii "KI-000\n"   // Повідомлення з символом нового рядка
```

Давайте розглянемо ключові моменти коду:

`.global _start` та `.align 2` - це директиви асемблера для налаштування програми.

Секція `.text` містить виконуваний код.

`mov` використовується для переміщення значень у регістри.

`adrp` та `add` разом завантажують повну адресу повідомлення, що є специфічним для macOS на ARM64.

`svc #0x80` викликає системний виклик, використовуючи номер, заданий у x16.

Секція `.data` містить повідомлення з номером групи "KI-000\n".

Цей код використовує системні виклики macOS для виведення рядка та завершення програми, що є типовим для низькорівневого програмування на Unix-подібних системах.

Для запуску програми необхідно скопіювати код у файл з назвою lab2.s, а потім виконати наступні кроки.

Зберіть (асемблюйте) код, використовуючи команду:

```
as -o lab2.o lab2.s
```

де:

- `as` - це асемблер
- `-o lab2.o` - це вихідний об’єктний файл
- `lab2.s` - вхідний файл з асемблерним кодом.

Створіть виконуваний файл, зв'язавши об'єктний файл за допомогою лінкера:

```
ld -o lab2 lab2.o -lSystem -syslibroot `xcrun -sdk macosx
--show-sdk-path` -e _start
```

де:

- `ld` - це лінкер
- `-o lab2` - вказує ім’я виконуваного файлу
- `lab2.o` - це вхідний об’єктний файл
- `-lSystem` - підключає системну бібліотеку
- `-syslibroot xcrun -sdk macosx --show-sdk-path` вказує шлях до SDK macOS
- `-e _start` вказує точку входу в програму.

Ці команди компілюють ваш асемблерний код у виконуваний файл для macOS на архітектурі ARM64. Після компіляції ви можете запустити виконуваний файл, щоб побачити результат виконання програми:

```
./lab2
```

Проаналізуйте отриманий результат.

1.2.4 Приклад програми для Linux з x86_64 чіпом

Наступним кроком давайте розглянемо приклад програми для Linux з x86_64 чіпом. Проаналізуйте код на Лістингу 1.3.

Лістинг 1.3 - Приклад програми для Linux з x86_64 чіпом

```
  global _start
section .text
_start:
    ; Підготовка до системного виклику write
    mov rax, 1          ; Номер системного виклику write
    mov rdi, 1          ; Файловий дескриптор (1 = stdout)
    mov rsi, message    ; Адреса повідомлення
    mov rdx, 7          ; Довжина повідомлення
```

```
    syscall             ; Виклик системного виклику
    ; Підготовка до системного виклику exit
    mov rax, 60         ; Номер системного виклику exit
    xor rdi, rdi        ; Статус виходу 0
    syscall             ; Виклик системного виклику
section .data
message: db "KI-000", 10  ; 10 - це символ нового рядка
```

Якщо проаналізувати код і порівняти його з минулим прикладом, можна побачити наступні відмінності:

- Використовуються регістри `rax`, `rdi`, `rsi`, `rdx` замість `x0`, `x1`, `x2`, `x16`
- Системні виклики мають інші номери (1 для write, 60 для exit)
- Використовується інструкція `syscall` замість `svc`

- Синтаксис директив трохи відрізняється (наприклад, `section` замість `.section`).

Також слід звернути увагу на директиву асемблера `db`,що означає "define byte" або "визначити байт". Вона використовується для визначення даних у пам'яті та вказує асемблеру, що потрібно розмістити дані в пам'яті побайтово. Отже, кожен символ у рядку "KI-000" автоматично перетворюється на відповідний ASCII-код і зберігається як окремий байт.

Число 10 в кінці - це ASCII-код для символу нового рядка ('\n'). Воно також зберігається як один байт.

`message:` - це мітка, яка вказує на початок цих даних у пам'яті.

Отже, ця інструкція створює в пам'яті послідовність байтів, що відповідає рядку "KI-000" з символом нового рядка в кінці. У програмі ми можемо звертатися до цих даних, використовуючи мітку `message`.

У різних асемблерах можуть використовуватися різні директиви для визначення даних. Наприклад, у синтаксисі AT&T, який часто використовується в Unix-подібних системах, аналогічна директива називається `.ascii` або `.string`.

Також ви можете звернути увагу на зміни у процесі компіляції. Для Linux з процесором x86_64 потрібно використовувати інший асемблер та лінкер. Зазвичай це NASM (Netwide Assembler) для асемблювання та ld для лінкування.

Збережіть код у файл `lab2.asm`, потім виконайте наступні команди:

Для асемблювання:

```
nasm -f elf64 -o lab2.o lab2.asm
```

Для лінкування:

```
ld -o lab2 lab2.o
```

Запуск програми:

```
./lab2
```

Основні відмінності в процесі компіляції полягають в наступному:

- Використовується `nams` замість `as`.
- Формат виводу - `elf64` для 64-бітних систем Linux. ELF (Executable and Linkable Format) - це формат виконуваного файла, об'єктних модулів, бібліотек та системних дампів.
- Команда лінкування простіша, не потребує додаткових опцій.

Ці зміни враховують різниці між архітектурами ARM64 та x86_64, а також між операційними системами macOS та Linux.

1.2.5 Приклад програми для Windows з x86_64 чіпом

Наступним кроком розглянемо приклад програми для Windows з x86_64 чіпом. Для створення програми виводу повідомлення на асемблері для Windows 10 або 11 з процесором x86_64 потрібно внести суттєві зміни як у код, так і в процес компіляції. Проаналізуйте код на Лістингу 1.4.

Лістинг 1.4 - Приклад програми для Windows з x86_64 чіпом

```
; Для 64-бітної версії Windows
extern ExitProcess
extern GetStdHandle
extern WriteConsoleA

section .data
    message db 'KI-000', 0Ah, 0

section .bss
    written resq 1

section .text
global main

main:
    ; Отримуємо дескриптор стандартного виводу
    mov rcx, -11                    ; STD_OUTPUT_HANDLE
    call GetStdHandle
```

```
    ; Зберігаємо дескриптор виводу
    mov r12, rax                    ; Зберігаємо результат
GetStdHandle в r12

    ; Підготовка параметрів для WriteConsoleA
    mov rcx, r12                    ; hConsoleOutput
    lea rdx, [message]              ; lpBuffer
    mov r8, 5                      ; nNumberOfCharsToWrite
    lea r9, [written]               ; lpNumberOfCharsWritten
    xor eax, eax                    ; Обнуляємо верхню частину
rax
    push rax                        ; lpReserved (must be 0)
    sub rsp, 32                     ; Виділяємо 32 байти
тіньового простору

    ; Виводимо повідомлення
    call WriteConsoleA

    ; Відновлюємо стек
    add rsp, 40                     ; 32 + 8 (для lpReserved)

    ; Виходимо з програми
    xor ecx, ecx                    ; exit code 0
    call ExitProcess
```

Основні відмінності, у порівнянні з попередніми операційними системами та архітектурами, полягають у наступному:

- Використовуються функції Windows API замість системних викликів Linux/macOS
- Синтаксис відрізняється (наприклад, `section` замість `.section`)
- Використовується `main` як точка входу замість `_start`
- Також є різниця в процесі компіляції.

Для Windows вам знадобиться інший набір інструментів. Зазвичай використовують NASM (Netwide Assembler) для асемблювання та LINK (з набору інструментів Microsoft Visual Studio) для лінкування.

Якщо ви користуєтеся операційною системою Windows, збережіть код у файл `lab2.asm`, потім виконайте наступні кроки:

Для асемблювання:

```
nasm -f win64 -o lab2.obj lab2.asm
```

Для лінкування:

```
link /subsystem:console lab2.obj kernel32.lib /entry:main
```

Запуск програми:

```
./lab2
```

Основні відмінності в процесі компіляції полягають в наступному:

- Використовується формат `win64` для 64-бітної версії Windows
- Команда лінкування використовує `link` замість `ld` і потребує додаткових параметрів
- Необхідно вказати підсистему (`/subsystem:console`) та точку входу (`/entry:main`)
- Підключається бібліотека `kernel32.lib` для доступу до функцій Windows API.

Також, для компіляції на Windows вам потрібно мати встановлені відповідні інструменти розробки, такі як Visual Studio або Windows SDK.

Ці зміни враховують відмінності між операційними системами (Windows замість macOS/Linux) та специфіку роботи з Windows API замість POSIX-сумісних системних викликів.

1.2.6 Компіляція та виконання програми

Після того, як ви проаналізували роботу всіх трьох прикладів коду, виберіть приклад програми для вашої операційної системи (у розділах 1.2.3 - 1.2.5), скопіюйте код, вкажіть в повідомленні вашу групу, скомпілюйте та запустіть програму. Зафіксуйте виконання всіх команд скриншотами у звіті. Також, додайте лістинг зміненої програми у звіт.

1.2.7 Внесення змін до програми

Після того, як ви виконали компіляцію і запуск програми для своєї операційної системи, вам необхідно внести додаткові зміни до програми. А саме, допишіть до повідомлення з номером групи ваше прізвище. Наприклад, “KI-221 Shevchenko”. Виконайте всі необхідні команди для перезапуску зміненої програми. Переконайтеся, що вивід команди вірний.

Зафіксуйте результат виконання команд скриншотом у звіті.

Створіть нову гілку в проєкті https://github.com/armymotion/cpnu_sysprog_2024 та додайте код зміненої програми, який має знаходитись в KI-000/nickname/lab2. Створіть Pull Request.

1.3 Вимоги до звіту

1.3.1 Основні вимоги до оформлення

Звіт оформлюється у вигляді текстового документа на сторінках формату А4 відповідно до вимог ДСТУ3008:2015, розділ 7 [2].

Шрифт (окрім лістингів коду) Times New Roman, 14, без курсиву і підкреслень. Для лістингів шрифт Courier New 12.

Поля: верхнє і нижнє від 1.5 до 2, ліве – 2.5, праве 1.5.

Нумерація сторінок праворуч, зверху.

Верхній колонтитул має містити прізвище, ім’я студента та групу.

Абзаци основного тексту вирівнюються по ширині. Перший рядок абзацу повинен мати відступ 1.25 або 1.3, між абзацами основного тексту інтервал не потрібен.

Текст має бути структурованим на розділи, підрозділи, пункти та, можливо, підпункти.

Для нумерації структурних елементів використовувати багаторівневий список. Після номера крапка не ставиться, роздільником має бути пробіл.

На рисунки, таблиці та лістинги в тексті мають бути посилання.

Рисунки, таблиці, лістинги нумеруються в межах розділу, тобто номер складається з двох чисел – номера розділу і порядкового номера рисунка з роздільником у вигляді крапки. Перед номером має бути слово «Рисунок » або «Таблиця », або «Лістинг » з пробілом. Після номера має стояти роздільник тире « – » і далі назва з великої літери.

Підпис рисунка розташовують під рисунком по центру. Назви таблиць і лістингів розташовують зверху, вирівнюють по лівому краю з абзацним відступом.

Рисунок має бути елементом тексту, вирівнюватися по центру, без абзацного відступу і мати інтервали відступу від попереднього тексту. Те саме стосується підпису під рисунком і назв таблиць і лістингів. Вони не повинні зливатися з основним текстом.

Заголовок розділу (назва лабораторної роботи) друкується великими жирними літерами і вирівнюється по центру без абзацного відступу.

Заголовки підрозділів, пунктів і підпунктів вирівнюються по лівому краю з абзацним відступом.

Усі заголовки повинні мати інтервал від попереднього тексту і перед наступним текстом.

1.3.2 Вміст звіту

Результатом лабораторної роботи є завантажений звіт. Рекомендований формат звіту є PDF-файл, в якому будуть відображені наступні результати роботи:

- Лістинг з кодом програми для вашої ОС і номером вашої групи

- Скриншоти виконаних команд асемблювання, компіляції та запуску програми з виводом
- Лістинг з кодом програми для вашої ОС, з номером групи і вашим прізвищем у виводі повідомлення
- Скриншоти виконаних команд асемблювання, компіляції та запуску програми з виводом після додавання прізвища
- Посилання на ваш Pull Request в репозиторії https://github.com/armymotion/cpnu_sysprog_2024 з кодом другої програми
- Висновки.