Зображення, що містить будівля, ескіз, малюнок, замок

Автоматично згенерований опис  
Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Кібербезпека та захист інформації:

Архітектура комп’ютерних систем

Студент: Бичок Вадим Вячеславович

Робота: 4

Назва: Основи побудови програми на асемблері в архітектурі x64 в операційній системі Windows з використанням WinAPI.

Використані програми: NASM, GoLink

Завдання:

1. Ознайомитись з теоретичними положеннями
2. Визначити змінні, занести відповідні значення у регістри та організувати цикл роботи для архітектури х64 на операційній системі Windows згідно свого варіанту
3. Визначити дані a(1) → 12, a(2) → 17, а(3) → 19, c1 → 15, c2 → 26 Занести в регістри такі величини RAX → (a(2) - a(1)) \* а(3), RBX → а(1) + а(2), RCX → c1 + c2, RDX → Організувати цикл, послідовно зменшуючи число у регістрі RСХ на 3. У циклі збільшувати число, що знаходиться у регістрі RВХ на величину, що знаходиться у регістрі RАХ, у тому випадку, коли значення у регістрі RСХ – непарне число, поки значення RСХ не стане менше -5.

Повідомляємо лінкеру GoLink що точка входу start.

global *start*

Імпортуємо функції, які будемо використовувати.

    extern *CreateFileA*

    extern *WriteFile*

    extern *CloseHandle*

    extern *ExitProcess*

    extern *GetLastError*

Ініціалізуємо змінні, які будемо використовувати.

section .data

*a:* DQ 12, 17, 19

*c1:* DQ 15

*c2:* DQ 26

*biff* DQ 0

*filename* DB 'output.txt', 0

*minus* DB '-', 0

*buffer* TIMES 16 DB 0

*bytes\_written* DQ 0

*error\_code* DQ 0

Вносимо значення в регістри, як сказано по завданню. Так як алгоритм схожий до минулих двох лабораторних робіт детально пояснювати цей пункт не будемо.

section .*text*

*start:*

Занесли значення а2 в регістр rax.

    MOV RAX, [*a*+8]

Відняли від rax(a2) значення а1.

    SUB RAX, [*a*]

Помножили різницю на а3.

    IMUL RAX, [*a*+16]

Занесли в регістр rbx а1 і потім додали до нього а2.

    MOV RBX, [*a*]

    ADD RBX, [*a*+8]

Аналогічно для rcx і с1 і с2.

    MOV RCX, [*c1*]

    ADD RCX, [*c2*]

Занесли а2 в rdx і виконали арифметичний зсув на 4 позиції ліворуч.

    MOV RDX, [*a*+8]

    SAL RDX, 4

Організовуємо цикл з самого початку перевіряємо чи значення в регістрі rcx не менше -5, якщо менше виходимо з циклу.

*loop1:*

    CMP RCX, -5

    JL *end*

Віднімаємо 3 від rax.

    SUB RCX, 3

Перевіряємо чи число парне, якщо парне, то повертаємося на початок циклу, якщо не парне то переходимо далі. Перевіряємо за допомогою TEST, він виконує побітове І, тобто кон’юнкцію, і якщо один з двох бітів 0, результат буде теж 0. Парне число завжди в кінці бітового рядка, його запис в двійковій системі, має 0, а непарне в кінці 1. Порівнюємо з 1 тому що якщо в кінці бітового рядка буде одиниця то кон’юнція дасть 1, а якщо в кінці 0, то кон’юнкція теж дасть 0, тобто покаже чи парне чи не парне число. Так як команда TEST після порівняння тільки встановлює флаг ZF і не змінює значення в регістрі, що нам і потрібно, тому це і використовуємо. Використовуємо умовний перехід JZ(Jump if Zero), щоб повернутися на початок циклу, тому що він відповідає за флаг ZF, який нам і потрібен.

    TEST RCX, 1

    JZ *loop1*

    JMP *ifnoeven*

Додаємо значення регістра RAX до RBX і повертаємося на початок циклу.

*ifnoeven:*

    ADD RBX, RAX

    JMP *loop1*

Ну і власне виведення в файл.

*end:*

Перетворюємо значення в регістрі rcx, яке потрібно вивести, на позитивне, так як функція переведення з бітового значення в рядок не вміє розрізняти додатні і від’ємні числа, і перетворює його на величезне число, тому просто зарання записали змінну, яка є рядком і має в собі символ «-», в кінці ми його допишемо перед основним числом.

neg rcx

    mov [*buffer*], rcx

Для переведення з бітового значення в рядок використовуємо надані в методиці функції.

*itoa:*

Зберігаємо значення rbp на стеку, щоб створити нову стекову рамку.  
 push rbp

Встановлюємо rbp як базовий вказівник поточної стекової рамки.  
 mov rbp, rsp

Виділяємо 8 байтів на стеку для тимчасових змінних.  
 sub rsp, 8

Копіюємо значення з rcx (початкове число) в rax.  
 mov rax, rcx

Встановлюємо rdi як вказівник на кінець буферу (місце для запису чисел у зворотному порядку).  
 lea rdi, [buffer+10]

Встановлюємо дільник для `idiv`, щоб виконувати ділення на 10.  
 mov rcx, 10

Ініціалізуємо лічильник символів у стеку (використовується для визначення довжини).  
 mov qword [rbp-8], 0

*divloop:*

Очищуємо rdx, щоб запобігти помилкам при діленні, оскільки idiv використовує rdx:rax як значення для ділення. Число для ділення (ділене): idiv використовує значення з регістрів RDX:RAX (або EDX:EAX для 32-бітних чисел). Це означає, що ділиться число, яке складається з об'єднаних регістрів RDX та RAX. У цьому "пара" регістрів розглядається як одне велике число..  
    xor rdx, rdx

Ділимо rdx:rax на rcx (10). Частка (результат) залишається в rax, залишок (остання цифра) в rdx.  
    idiv rcx

Перетворюємо цифру з rdx в ASCII-код, додаючи 0x30 (ASCII-код символу '0').  
    add rdx, 0x30

Зсуваємо rdi ліворуч, щоб записати наступну цифру(віднімаємо 1 від адреси, щоб записати наступні цифри як десятки сотні і тд.).  
    dec rdi

Записуємо обчислену цифру як ASCII-символ у buffer.  
    mov byte [rdi], dl

Збільшуємо лічильник символів, щоб знати довжину результату(додаємо 1 до значення яке лежить за адресою [rbp-8]).  
    inc qword [rbp-8]

Перевіряємо, чи частка не дорівнює нулю. Якщо ні, повторюємо цикл.  
    cmp rax, 0

    jnz *divloop*

Результат функції — адреса початку рядка з числом — записується в rax.  
    mov rax, rdi

Зберігаємо адресу початку рядка в buffer.  
    mov [*buffer*], rdi

Перевіряємо чи все працює правильно.  
    mov rax, [*buffer*]

Відновлюємо значення rbp і rsp, щоб завершити функцію.  
    leave

Виділяємо 56 байтів на для стеку.  
    sub rsp, 56

Заносимо аргументи функції у правильному порядку. RCX – перший аргумент, для функції CreateFileA це ім’я файлу який потрібно або відкрити або створити, тут це адреса змінної, в якій лежить назва файлу.  
    mov rcx, *filename*

RDX це другий аргумент функції, наразі це число яке означає з якими правами потрібно відкрити або створити файл, у нас це читання та запис.  
    mov rdx, 0C0000000h

R8 це третій аргумент функції, наразі це чи потрібен спільний доступ до файлу, 0 бо нам не потрібен  
    mov r8, 0

R9 це четвертий аргумент функції, наразі це чи потрібні якісь умови безпеки файлу, 0 бо нам не потрібно ускладнювати собі життя.  
    mov r9, 0

Далі аргументи функції записуються у стек, за конкретним адресом, по іншорму не буде працювати, функція тільки ці адреси сприймає як аргументи. За адресою [rsp + 32] повинен лежати 5 аргумент для функції, наразі це що робити якщо файл існує, або не існує, 2 бо ми хочемо постійно створювати новий файл, навіть, якщо він існує, його перестворимо.  
    mov r10, 2

    mov [rsp + 32], r10

За адресою [rsp + 32] повинен лежати 6 аргумент для функції, наразі це які атрибути мають бути у файла, 128 – це стандартні атрибути, що нам і потрібно.  
    mov r10, 128

    mov [rsp + 40], r10

Заносимо в останній аргрумент функції значення регістра R9 так як там 0, а це опціональний аргумент, і він нам не потрібен.  
    mov [rsp + 48], r9

Викликаємо функцію CreateFaileA з уже заданими аргументами.  
    call *CreateFileA*

Вирівнюємо стек.  
    add rsp, 56

Перевіряємо чи успішно виконалася функція, якщо ні, то в регістр rax записується -1, тобто помилка, і ідемо до мітки отримання останньої помилки.  
    cmp rax, -1

    je *get\_error*

Почнемо з середини обнуляємо ці регістри, R9 залишиться 0.  
    mov r8, 0

    mov r9, 0

Після виконання попередньої функції в регістр був записаний дескриптор(адреса по якій зараз відкритий файл) записуємо її першим аргументом, відповідно до вимог функції.  
    mov rcx, rax

Зберігаємо дескриптор у змінній *biff* щоб потім ще була можливість записати ще щось у даний файл, по цьому дескриптору.  
    mov [*biff*], rax

Записуємо у другий аргумент функції адресу змінної значення якої нам потрібно записати, перше що ми хочемо записати це «-» а потім вже позитивне значення регістра RCX.  
    lea rdx, [*minus*]

Третій аргумент функції це кількість байтів яку потрібно записати, так як в нас один символ, тобто один байт, тому і і записуємо тільки один байт.  
    mov r8, 1

Викликаємо саму функцію, з уже заданими аргумментами.  
    call *WriteFile*

Перевіряємо на помилку.  
    cmp rax, -1

    je *get\_error*

Аналогічно робимо для значення регістру RCX, тільки функція яка переводила значення в рядок створює «мусор» за адресою нашого рядка, тому, щоб виводилося все гарно, знаючи всі нюанси регугулюємо виведення. Ще на ініціалізації змінних, був використаний трюк з TIMES ми задали 16 байтів змінній, щоб вона не залазила на інші змінні.  
    mov r8, 0

    mov r9, 0

    mov rcx, [*biff*]

    lea rdx, [*buffer*+9]

    mov r8, 1

    call *WriteFile*

    cmp rax, -1

    je *get\_error*

Виконуємо вихід з файлу за дескриптором.  
    mov rdi, rax

    call *CloseHandle*

Виконуємо вихід з програми. *exit\_program:*

    xor rcx, rcx

    call *ExitProcess*

Якщо були помилки то виводимо їх. *get\_error:*

    call *GetLastError*

    mov [*error\_code*], rax

    jmp *exit\_program*

Контрольні запитання:

1. Що таке WinAPI?  
   Windows Api (application programming interfaces) — загальне найменування для цілого набору базових функцій інтерфейсів програмування застосунків операційних систем сімейства Windows корпорації Майкрософт. Він є найпрямішим способом взаємодії застосунків з Windows.   
   Для створення програм, що використовують Windows API, Майкрософт випускає SDK, який називається Platform SDK і містить документацію, набір бібліотек, утиліт і інших інструментальних засобів.  
     
   Windows API був спочатку спроєктований для використання в програмах, написаних на мові C (або C++). Робота через Windows API — це найближчий до системи спосіб взаємодії з нею прикладних програм. Нижчий рівень доступу, необхідніший тільки для драйверів пристроїв, в поточних версіях Windows надається через Windows Driver Model.
2. Що таке Shadow Space?  
   Shadow Space — це виділена область в стеку, яка резервується перед викликом функцій. Ця область дає змогу функції гарантовано використовувати простір для параметрів без необхідності додатково змінювати стек.  
     
   **Розмір**: У x64-архітектурі Windows зарезервований простір становить 32 байти (або 4 слоти по 8 байт). Цей розмір визначений стандартом ABI, і всі функції зазвичай дотримуються цього правила.  
     
   Хоча аргументи зазвичай передаються через регістри (RCX, RDX, R8, R9), кожна функція виділяє Shadow Space для можливості зберігати ці параметри на стеку.

Цей простір надає зручне місце для зберігання копій параметрів або для використання під внутрішні змінні функції, якщо це необхідно.  
  
Shadow Space розташовується на початку стеку після створення стек-фрейму, і, зазвичай, в ньому зберігаються перші чотири параметри функції.  
  
Оскільки простір є зарезервованим, він завжди доступний, що робить виклики функцій більш ефективними і стандартизованими.

1. Особливості регістрів у x64.  
   Збільшена розрядність:   
   Регістри розширені з 32 до 64 бітів. Для цього використовуються префікси R замість E для x86-версії. Наприклад, 64-бітний регістр RAX замінює 32-бітний EAX.  
     
   Однак доступні і 32-бітні частини 64-бітних регістрів, а також менші частини (16-бітні та 8-бітні), як-от AX, AH, AL.  
     
   Додаткові регістри:  
   У x64 доступно 16 загальних регістрів, на відміну від x86, де було 8. Нові регістри (R8–R15) надають більше місця для зберігання проміжних результатів і аргументів функцій.  
     
   Ці додаткові регістри (R8-R15) також мають 8-бітні (R8B-R15B), 16-бітні (R8W-R15W) і 32-бітні (R8D-R15D) версії для зручнішого доступу до меншого обсягу даних.  
     
   Керування пам’яттю і стеком:  
   Регістр RSP (Stack Pointer) вказує на поточну позицію в стеку. Він автоматично оновлюється при викликах і поверненнях із функцій.  
     
   RBP (Base Pointer) часто використовується як фіксатор бази для стек-фреймів, хоча в x64 його використання стало менш необхідним через оптимізацію компілятора (багато програм обходяться без RBP, використовуючи RSP).  
     
   Спеціальні регістри:  
   RAX: використовується для збереження результатів арифметичних операцій і повернення значень із функцій.  
   RBX, RBP: базові регістри, часто використовуються для зберігання даних і не змінюються під час викликів функцій (callee-saved).  
   RSI та RDI: джерело (Source Index) і ціль (Destination Index) для операцій роботи з пам'яттю, таких як movsb.  
   RIP: інструкційний покажчик (Instruction Pointer), що містить адресу наступної команди для виконання. RIP-відносна адресація дозволяє створювати позиційно-незалежний код.  
   RFLAGS: регістр прапорів, що зберігає результат виконання інструкцій і використовується умовними переходами.  
     
   Callee-saved та Caller-saved регістри:  
   **Caller-saved**: регістри (RAX, RCX, RDX, R8-R11), які викликач має зберегти, якщо потрібні значення в них після виклику функції.  
   **Callee-saved**: регістри (RBX, RBP, R12-R15), які функція має зберегти і відновити до завершення.
2. Поясніть особливість роботи зі стеком та наповнення регістрів у Windows.  
   Windows використовує x64 calling convention, яка задає порядок використання стеку і регістрів для передачі параметрів функціям, а також управління пам’яттю для стеку. Це конвенція для 64-бітної архітектури Windows, відома як Microsoft x64 calling convention.  
     
     
   Передача аргументів через регістри  
   Перші **чотири параметри** функції передаються в регістри **RCX, RDX, R8, і R9** відповідно (якщо це цілочислові значення або вказівники). Додаткові аргументи (якщо їх більше чотирьох) передаються через стек. Це дозволяє уникнути зайвих операцій з пам’яттю і пришвидшує виконання, оскільки регістри використовуються швидше, ніж звернення до стеку.  
     
   Shadow Space (або "запас для аргументів")  
   Для кожного виклику функції резервується **Shadow Space** (або "примарний простір") — 32 байти на стеку.  
   Shadow Space виділяється викликачем перед викликом функції і звільняється після завершення.  
     
   Caller-saved і Callee-saved регістри  
     
   Повернення значень  
   Значення, яке повертає функція, зазвичай розміщується у **регістрові RAX**.
3. Особливості використання запису у файл з використанням WinAPI.  
   Розписано у протоколі, вище під час виконання завдання.
4. Особливості використання MessageBox з використанням WinAPI.  
   Має всього 4 аргументи, для яких вистачає регістрів, тому потрібно просто в правильній послідовності занести аргументи, вони ¾ аргументів не обов’язкові, що ще більше полегшує використання функції.
5. У яких регістрах зберігаються значення з плавочою точкою при використанні WinAPI?  
   Значення з плаваючою точкою зберігаються в регістрах SIMD (Streaming SIMD Extensions), тобто у **XMM0 - XMM15**. Це специфічні регістри, призначені для роботи з числами з плаваючою точкою в 64-бітній архітектурі Windows.
6. Поясніть різницю між х64 та х32 архітектурами.  
   Збільшена розрядність:   
   Регістри розширені з 32 до 64 бітів. Для цього використовуються префікси R замість E для x86-версії. Наприклад, 64-бітний регістр RAX замінює 32-бітний EAX.  
     
   Однак доступні і 32-бітні частини 64-бітних регістрів, а також менші частини (16-бітні та 8-бітні), як-от AX, AH, AL.  
     
   Додаткові регістри:  
   У x64 доступно 16 загальних регістрів, на відміну від x86, де було 8. Нові регістри (R8–R15) надають більше місця для зберігання проміжних результатів і аргументів функцій.  
     
   Керування пам’яттю і стеком:  
   Регістр RSP (Stack Pointer) вказує на поточну позицію в стеку. Він автоматично оновлюється при викликах і поверненнях із функцій.  
     
   RBP (Base Pointer) часто використовується як фіксатор бази для стек-фреймів, хоча в x64 його використання стало менш необхідним через оптимізацію компілятора (багато програм обходяться без RBP, використовуючи RSP).