МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

СИСТЕМНЕ ПРОГРАМУВАННЯ. ПРОГРАМУВАННЯ НА АСЕМБЛЕРІ

Комп'ютерний практикум

Навчальний посібник

Укладач: В. М. Порєв

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Комп’ютерні системи та мережі»
спеціальності 123 «Комп’ютерна інженерія»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Системне програмування. Програмування на асемблері: комп'ютерний практикум [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Комп'ютерні системи та мережі» спеціальності 123 «Комп'ютерна інженерія» / КПІ ім. Ігоря Сікорського; уклад. Порєв В.М. – Електронні текстові дані (1 файл: 3,2 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 146 с. – Назва з екрана.

Гриф надано Методичною радою КПІ м.. Ігоря Сікорського
(протокол № 1 від 02.09.2022 р.)
за поданням Вченої ради факультету інформатики та обчислювальної техніки
(протокол № 11 від 11.07.2022 р.)

Електронне мережне навчальне видання

Укладач: Порєв Віктор Миколайович, к.т.н.

Рецензент: Тарасенко-Клятченко О.В., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет прикладної математики, кафедра системного програмування і спеціалізованих комп'ютерних систем

Відповідальний редактор Волокита А.М., к.т.н., доцент, КПІ ім. Ігоря Сікорського, факультет інформатики та обчислювальної техніки, кафедра обчислювальної техніки

У навчальному посібнику викладені теоретичні положення та практичні завдання для виконання комп'ютерного лабораторного практикуму при вивченні дисципліни «Системне програмування». Наведено методичні рекомендації для використання асемблера при роботи з проєктами у середовищі Microsoft Visual Studio та інших інструментальних засобів. Приділено увагу аспектам розроблення на асемблері модулів для проектів програмних додатків.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою за освітньою програмою «Комп’ютерні системи та мережі» спеціальності 123 «Комп’ютерна інженерія» Може бути також корисним студентам, які вивчають системне програмування у курсах споріднених дисциплін відповідних спеціальностей.

Реєстр. № НП 22/23-014.Обсяг 4,4 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© КПІ ім. Ігоря Сікорського, 2022

ЗМІСТ

ВСТУП 5

Лабораторна робота №1. Знайомство із засобами розробки програм на асемблері 6
Завдання 6
Теоретичні відомості 6
Методичні рекомендації 20
Варіанти завдань та основні вимоги 39
Контрольні запитання 40

Лабораторна робота №2. Створення модульних проектів на Асемблері та вивчення форматів представлення чисел 41
Завдання 41
Теоретичні відомості 41
Методичні рекомендації 46
Варіанти завдань та основні вимоги 50
Контрольні запитання 52

Лабораторна робота №3. Програмування арифметичних операцій підвищеної розрядності 53
Завдання 53
Теоретичні відомості 53
Методичні рекомендації 60
Варіанти завдань та основні вимоги 61
Контрольні запитання 63

Лабораторна робота №4. Програмування множення чисел підвищеної розрядності 64
Завдання 64
Теоретичні відомості 64
Методичні рекомендації 69
Варіанти завдань та основні вимоги 70
Контрольні запитання 71

Лабораторна робота №5. Програмування побітових операцій 72
Завдання 72
Теоретичні відомості 72
Методичні рекомендації 81
Варіанти завдань та основні вимоги 82
Контрольні запитання 84

Лабораторна робота №6. Програмування операцій ділення чисел 85
Завдання 85
Теоретичні відомості 85
Методичні рекомендації.. 96
Варіанти завдань та основні вимоги 97
Контрольні запитання 99

Лабораторна робота №7. Виконання операцій з плаваючою точкою та вивчення команд x87 FPU 100
Завдання 100
Теоретичні відомості 100
Методичні рекомендації... 107
Варіанти завдань та основні вимоги 111
Контрольні запитання 112

Лабораторна робота №8. Використання системних функцій у програмах на асемблері 113
Завдання 113
Теоретичні відомості 113
Методичні рекомендації... 117
Варіанти завдань та основні вимоги 121
Контрольні запитання 122

Лабораторна робота №9. Використання у проекті C++ модулів на асемблері 123
Завдання 123
Теоретичні положення 124
Методичні рекомендації... 127
Варіанти завдань та основні вимоги 144
Контрольні запитання 145

РЕКОМЕНДОВАНА ЛІТЕРАТУРА 146

ВСТУП

Для виконання циклу робіт комп’ютерного лабораторного практикуму рекомендовано технологію Microsoft Visual Studio. У комп'ютерному класі вона розгорнута у операційному середовищі Windows. Але для виконання завдань та проведення експериментів студенти можуть використовувати власні комп'ютери вдома. Особливо це стосується організації занять дистанційно. За погодженням з викладачем студенти можуть використовувати й інші технологічні засоби створення програм. Так, зокрема, для вирішення завдань (з певною адаптацією цих завдань) можливо використати MASM32, MASM64, FASM, NASM, OllyDbg тощо. Також можливо обрати й інше операційне середовище, зокрема, Linux

Основна частина лабораторних робіт присвячена розробці програмних за стосунків у вигляді проектів самодостатніх додатків для Windows. Остання лабораторна робота №9 присвячена розробці рішення, у якому поєднуються модулі на асемблері в проекті на C++.

Для заочної форми навчання перелік обов’язкових до виконання робіт може бути скороченим до № 1, 3, 4 та 9.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітньою програмою за освітньою програмою «Комп’ютерні системи та мережі» спеціальності 123 «Комп’ютерна інженерія» Може бути також корисним студентам, які вивчають системне програмування у курсах споріднених дисциплін відповідних спеціальностей.

Лабораторна робота №1. Знайомство із засобами розробки програм на асемблері

Мета: навчитися створювати проекти програм на асемблері у середовищах розробки програмного забезпечення та отримати перші навички налагодження програм.

Завдання

1. Інсталювати програмний пакет MASM32. Написати вихідний текст найпростішої програми **Lab1** на асемблері. Скомпілювати вихідний текст і отримати виконуваний файл програми. Перевірити роботу програми.
2. Інсталювати Microsoft Visual Studio. Створити у середовищі MS Visual Studio проект з ім'ям **Lab1_cpuid**. Встановити необхідні параметри проекту – опції середовища розробки програм. Написати вихідний текст програми згідно з варіантом завдання. Скомпілювати вихідний текст і отримати виконуваний файл програми. Перевірити та налагодити програму. Отримати дизасемблерний текст машинного коду і проаналізувати його.
3. У звіті по лабораторній роботі надати опис програм **Lab1** та **Lab1_cpuid**.

Теоретичні відомості

Асемблер – це мова програмування, яка максимально наближена до машинної мови цифрового комп'ютера.

Формат рядків вихідного тексту

Текст програми на мові асемблера складається з багатьох рядків, у кожному рядку загалом можуть бути записані такі елементи: мітка, команда або директива, коментар:

```
[мітка] [команда] [;коментар]
```

або

```
[мітка] [директива] [;коментар]
```

Квадратні дужки позначають необов'язкові елементи. Будь-який елемент є необов'язковим – можуть бути і порожні рядки (це

використовується для покращення сприйняття людиною окремих блоків тексту). Порожні рядки асемблер ігнорує.

Мітка – складається з букв англійського алфавіту, цифр та символів _, @ (окрім вказаних, для мітки можуть використовуватися й інші символи, проте це не рекомендується). Мітка не повинна починатися з цифри.

Коментар – це текст пояснень, починається з символу '; ' і містить будь-який набір символів до кінця рядка. Коментар асемблер ігнорує.

Скелет вихідного тексту програми

Текст програми складається з набору базових блоків тексту, які виділяються директивами асемблера. Запис директив, їхній набір та синтаксис можуть бути різним – це залежить, по-перше від типу та версії асемблера, а по-друге типом програми (або програмного модуля) і середовищем, у якому повинна виконуватися програма – типом процесора, режимом роботи процесора та операційною системою.

Узагальнено скелет вихідного тексту можна представити як три обов’язкові директиви, які повинні бути завжди, та решту частину тексту:

```
Директива, що визначає тип процесора
Директива, що визначає модель памяті
Решта частина тексту
Директива END
```

Розглянемо скелет тексту програми Win32 – звичайної програми, розробленої для роботи у середовищі 32-бітної операційної системи Windows. Така програма впродовж роботи буде викликати різноманітні системні функції. Зокрема, будь-яка програма для коректного завершення своєї роботи обов’язково повинна викликати функцію ExitProcess системної бібліотеки kernel32. Крім того, для програм Win32 типовим є використання елементів графічного інтерфейсу користувача (бібліотека user32) та графіки (gdi32).

```
.386
.model flat, stdcall
option casemap :none  ; розрізнювати великі та маленькі букви
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
```

```
include \masm32\include\gdi32.inc
;  а також include для інших заголовочних файлів

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
;  а також includelib для інших бібліотек

.const
;  оголошення та ініціалізація констант

.data
;  оголошення статичних даних (наприклад, глобальних перемінних)

.code
start:


;  програмний код

  invoke ExitProcess, 0     ;  завершення роботи програми
end start
```

Директива .386 означає, що програма призначається для виконання на процесорах типу Intel 80386 і вище.

Директивою .MODEL вказана модель адресації пам'яті **flat** та спосіб (конвенція) виклику **stdcall**.

Точка входу у програму позначається міткою, ім'я якої потім вказане у директиві END – у наведеному вище тексті це ім'я "start". Точкою входу буде перша команда за цією міткою.

Директива INVOKE асемблера MASM означає виклик процедури. Її зручно використовувати замість традиційного **call** для виклику процедур з параметрами, які передаються через стек командами **push**. Наприклад, замість

```
push style
push offset Caption
push offset Text
push hWnd
call MessageBoxA
```

зручніше записати

```
invoke MessageBoxA, hWnd, ADDR Text, ADDR Caption, style
```

що робить синтаксис інструкції дещо подібним до мови високого рівня.

Наприкінці роботи програма викликає системну функцію **ExitProcess** з параметром (аргументом) нуль – нульове значення буде передане операційній системі як код завершення процесу. Зазвичай прийнято, що нульове значення означає нормальне завершення процесу.

Директиви створення даних

У програмах часто використовуються дані у вигляді перемінних та констант. Для створення таких даних у асемблері є директиви:

DB (*define byte*) – визначає перемінну розміром у 1 байт;
DW (*define word*) – визначає перемінну розміром у 2 байти (слово);
DD (*define double word*) – визначає перемінну розміром у 4 байти (подвійне слово);
DQ (*define quad word*) – визначає перемінну розміром у 8 байтів (квадрослово);
DT (*define ten bytes*) – визначає перемінну розміром у 10 байтів.

Формат запису директиви для створення перемінної або константи:

```
<ім'я> Dx <операнд> [, <операнд>]
```

де позначка Dx означає одну з директив DB, DW, DD, DF, DQ або DT. Усі ці директиви можуть бути використані як для створення простих перемінних та констант, так і для створення масивів. Масиви для рядків символів зазвичай створюють директивою DB (якщо у символів однобайтове кодування).

Створення перемінної може бути суміщено із початковою ініціалізацією деяким значенням – числом, символом, рядком тексту. Знак питання '?' означає без початкової ініціалізації. Якщо для ініціалізації вказується рядок або якщо записані декілька значень через кому, то пам'ять виділяється для декількох перемінних вказаного типу, тоді буде масив. Щоб при ініціалізації масивів не вказувати багаторазово одне значення, записують DUP (значення). Приклади створення даних надані нижче:

```
a dd ?              ; перемінна розміром 4 байти, неініціалізована
b dd 1.25           ; перемінна 4-байтова, ініціалізована числом 1.25 з плаваючою точкою
```

```
c db 10011001b      ; перемінна розміром 1 байт, ініціалізована двійковим числом
d db '!'            ; перемінна розміром 1 байт, ініціалізована символом '!'
e dw 1234h          ; перемінна розміром 2 байти, ініціалізована шістнадцятьковим числом
f db 'рядок',0            ; рядок з 6 байтів, завершується нулем (як прийнято у C, C++)
g db 'рядок',13,10          ; рядок з 7 байтів, завершується символами CR (13), LF (10)
h db 'рядок1',13,10,'рядок2',0      ; символи у двох рядках, завершуються нулем
k dd 100 dup (?)                    ; масив із 100 подвійних слів, неініціалізований
m dd 100 dup (0)                    ; масив із 100 подвійних слів, у які записано 0
n dq 1.25           ; перемінна 8-байтова, ініціалізована числом 1.25 з плаваючою точкою
p dq 10 dup (0,1,2) ; масив з 30 квадрослів – 10 трійок, ініціалізованих значеннями 0, 1, 2
q dt 1.25           ; перемінна 10-байтова, ініціалізована числом 1.25 з плаваючою точкою
```

Для перемінних та констант можна застосувати операції **offset** і **type**. Операція **offset** знаходить адресу розташування у пам’яті, **type** – розмір.

Команда MOV

Ця команда часто використовується у програмах на асемблері. Вона виконує копіювання даних. Команда MOV має два операнди:

```
mov Куди, Джерело
```

Перший операнд вказує, куди записати інформацію. Це може бути регістр або комірка пам'яті. У документації Intel це позначається як r/m.

Другий операнд – **Джерело** повинен вказувати, звідки взяти інформацію. Це може бути або регістр, або комірка памяті або безпосередньо числове значення (згідно документації – r/m/imm). Наприклад, запис

```
mov ecx, eax
```

означає скопіювати дані з регістру EAX у регістр ECX. У наступному рядку

```
mov ax, 5
```

запрограмований запис числа 5 у регістр AX. У якості другого операнду записане безпосередньо числове значення. Такі значення зберігаються у пам’яті, тому фактично виконується копіювання типу "пам’ять → регістр".

Певна кількість байтів джерела повинна записуватися у відповідне за розміром місце. Приклади помилкових записів:

```
mov al, edx        ; помилка: з 32-бітового у 8-бітовий регістр
mov ax, ecx        ; помилка: з 32-бітового у 16-бітовий регістр
mov eax, cx        ; помилка: з 16-бітового у 32-бітовий регістр
```

У наступних прикладах

```
mov al, 5      ; AL = 00000101 (8 біт)
mov ax, 5      ; AX = 0000000000000101 (16 біт)
mov eax, 5     ; EAX = 00000000000000000000000000000101 (32 біти)
```

помилок немає – у регістри записується відповідна кількість бітів значення, яке може представлятися як 8-бітовим, так і 16-, або 32-бітовим кодом.

У той же час запис

```
mov al, 259    ; помилка: у 8-бітовий регістр записати число 259 не можна
```

є неприпустимим, тому що для представлення числа 259 восьми бітів замало. Асемблер при компіляції видасть помилку.

Можна сказати, що запис

```
mov a, b
```

означає те саме, що запис a = b у мові програмування високого рівня. Розглянемо присвоєння значення одної перемінної іншій, запрограмоване мовою C/C++:

```
long a, b;

a = b;
```

Записати відповідний код на асемблері можна спробувати так:

```
.data
  a dd ?         ; створення неініціалізованої перемінної a
  b dd ?         ; створення неініціалізованої перемінної b

.code
mov a, b         ; помилка, так не можна
```

Перемінні a та b є об’єктами у пам’яті. Одною командою MOV копіювати з пам’яті у пам’ять не можна (у документації Intel немає варіанту mov m, m). Компілятор видасть повідомлення про помилку. А можна, наприклад, так:

```
mov eax, b       ; регістр EAX у якості посередника
mov a, eax
```

Необхідно зазначити, що у наведених вище рядках використання імен перемінних (a та b) у якості операндів команд MOV дещо спрощує синтаксис, проте ховає те, що насправді замість імен у коді використовуються адреси відповідних перемінних. Більш адекватний програмний код того, що насправді виконується, можна побачити у вікні дизасемблера.

```
mov eax, dword ptr [b]
mov dword ptr [a], eax
```

У мові асемблера квадратні дужки означають, що всередині них міститься вказівник – той, хто зберігає адресу пам’яті.

Синтаксис мови асемблера достатньо гнучкий. Зокрема, запис

```
 mov eax, b
```

можна вважати дещо спрощеним варіантом, аніж

```
 mov eax, [b]
```

який, у свою чергу, є спрощенням від:

```
 mov eax, dword ptr [b]
```

Остання форма запису є найбільш коректною та повною.

Розглянемо наступний приклад. Створимо масив з чотирьох 32-бітових елементів – елементів типу DWORD. Нехай і’мя масиву буде M

```
.data

  M dd 4 dup(?)
```

Елементи масиву розташовуються у пам’яті послідовно. Загалом для масиву M потрібно 16 байтів пам’яті. Адреса масиву вказує на його молодший байт

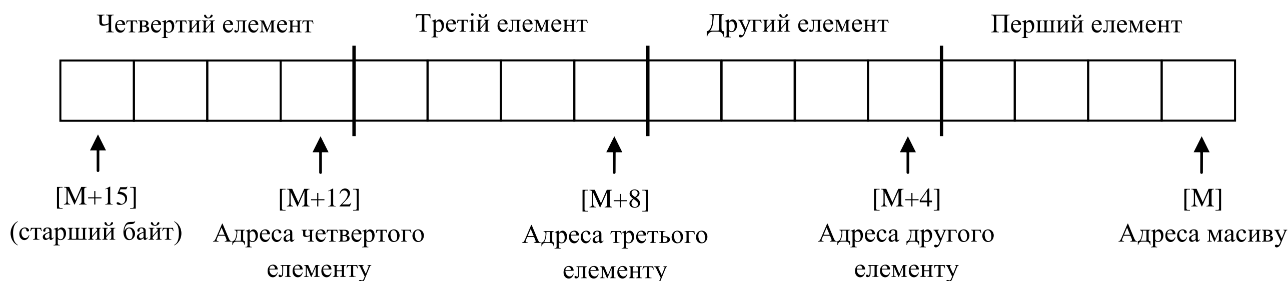


Рис. 1.1. Масив з чотирьох елементів типу DWORD

Запишемо у другий елемент якесь значення, наприклад:

```
mov dword ptr[M+4], 89ABCDEFh
```

Результат на рис. 1.2.

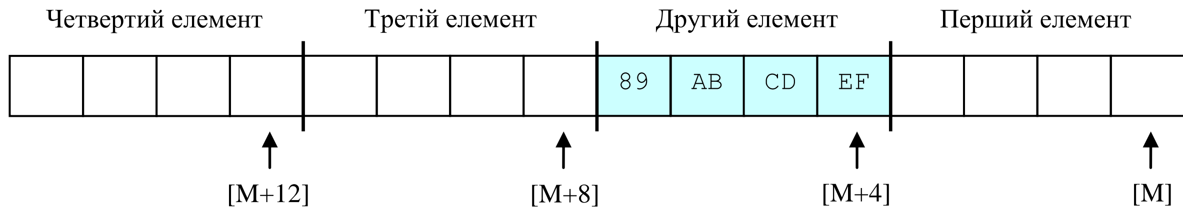


Рис. 1.2. Результат операції

Ще приклад:

```
mov dword ptr[M+7], 10CCABBAh
```

Результат на рис. 1.3

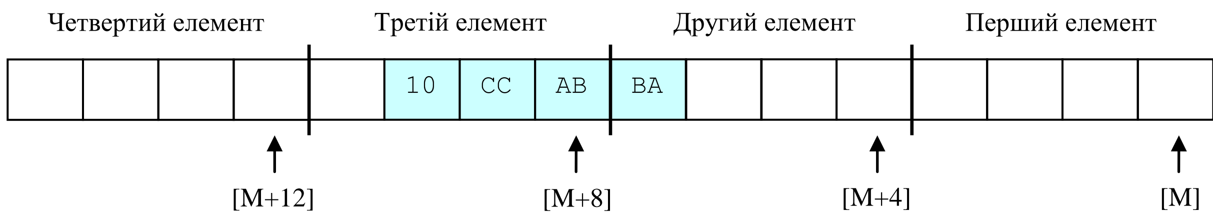


Рис. 1.3. Результат операції

Приклад запису двохбайтового значення у пам'ять:

```
mov word ptr[M+8], 4352h
```

Результат на рис. 1.4

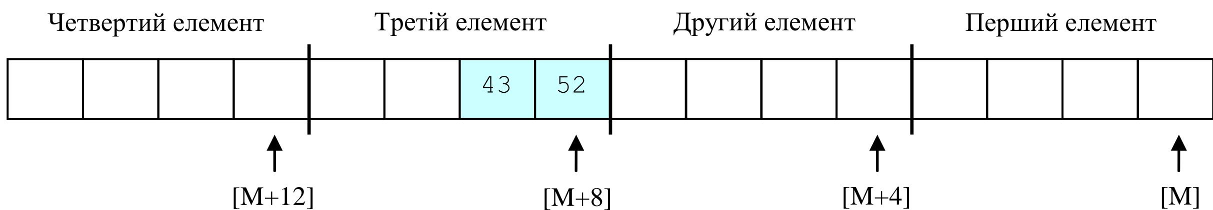


Рис. 1.4. Результат запису двохбайтового значення

Приклад запису однобайтового значення у пам'ять:

```
mov byte ptr[M+9], 2Ah
```

Результат на рис. 1.5:

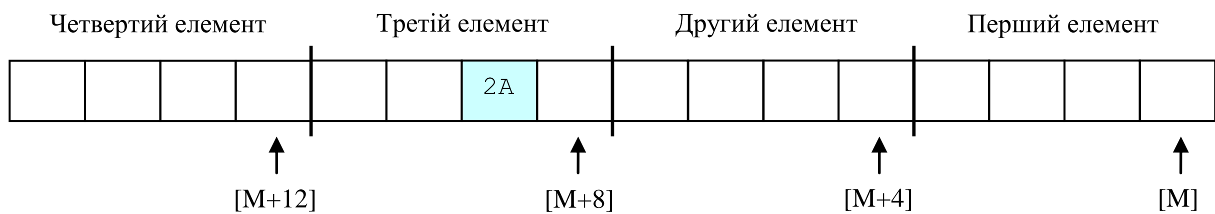


Рис. 1.5. Результат запису однобайтового значення

Команда CPUID

Ця команда призначена для ідентифікації процесора та отримання відомостей про його властивості. Для того, щоб вказати, які саме відомості потрібні, треба записати у регістр EAX деяке значення – параметр для команди. Результати роботи команди CPUID процесор записує у регістри EAX, EBX, ECX EDX.

Для того, щоб правильно користуватися командою CPUID, потрібно дотримуватися певної послідовності роботи програми. Можна спочатку перевірити – а чи можливо взагалі користуватися цією командою? Для цього треба перевірити значення 21-го біту регістру EFLAGS. Проте, у цій лабораторній роботі можна вважати це зайвим – маємо використовувати комп'ютери з процесорами, які гарантовано підтримують CPUID.

Отримання відомостей за допомогою команди CPUID робиться у декілька кроків. Спочатку треба виконати цю команду із значенням параметру 0:

```
mov eax,0

cpuid
```

Після виконання такого коду, у регістрі EAX міститься значення, яке означає максимально можливе значення параметру, яке можна використати для отримання базових відомостей. Якщо намагатися викликати команду CPUID із параметром більше зазначеного вище максимального, то результатом будуть усі нулі (це не стосується параметрів так званих розширених відомостей).

Після виконання команди CPUID із параметром 0 процесор також записує у регістри EBX, ECX та EDX коди символів імені процесора. Для

процесорів Intel це буде "GenuineIntel", для процесорів AMD – "AuthenticAMD" тощо. Необхідно відзначити, що 12 символів записуються четвірками у такому порядку – спочатку EBX, потім EDX, останні чотири у регістрі ECX.

Наступним кроком у програмі буде виконання команди з параметром 1:

```
mov eax,1
cpuid
```

Після виконання цієї команди у регістри EAX, EBX, ECX та EDX буде записано інформацію про сімейство процесорів, модель та деякі інші відомості.

Наступним кроком у програмі буде виконання команди з параметром 2:

```
mov eax,2
cpuid
```

І так далі, наскільки можливо. Як вже вказувалося вище, максимально можливе значення параметру стає відомим після виконання команди з параметром 0. Це для базового набору відомостей. Проте, є ще так звані, розширені відомості, які відповідають функціям на основі команди CPUID з параметром 80000000h і більше. Для того, щоб дізнатися максимальне значення для параметру розширених функцій, потрібно виконати:

```
mov eax,80000000h
cpuid
```

У результаті виконання цього у регістрі EAX буде деяке значення, наприклад, 80000008h. Можна послідовно виконувати команди CPUID із значеннями параметрів від 80000001h до 80000008h для отримання відповідних відомостей. Наприклад, після виконання коду

```
mov eax,80000008h
cpuid
```

у регістрі EAX буде інформація про максимальну можливу розрядність адрес пам’яті даного процесора.

Вичерпна інформація щодо команди CPUID міститься у документі "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference" [1].

У підсумку, виконання ланцюжка команд CPUID можна відобразити наступним чином

```
;---базові функції---
mov eax,0       ;початок
cpuid
. . .           ;зберігання значень EAX, EBX, ECX, EDX
mov eax,1
cpuid
. . .           ;зберігання значень EAX, EBX, ECX, EDX
. . .           ;якщо можливо, то інші функції для EAX>1

;---розширені  функції---
mov eax,80000000h
cpuid
. . .           ;зберігання значень EAX, EBX, ECX, EDX
mov eax,80000001h
cpuid
. . .           ;зберігання значень EAX, EBX, ECX, EDX
. . .           ;якщо можливо, то інші функції для EAX>80000001h
```

Для зберігання значень регістрів EAX, EBX, ECX та EDX можна їх записувати у масив-вектор, наприклад:

```
.data
  res dd 256 dup(0)

.code
  mov eax, 0
  cpuid
  mov dword ptr[res], eax
  mov dword ptr[res+4], ebx
  mov dword ptr[res+8], ecx
  mov dword ptr[res+12], edx
  . . .           ;у подібний спосіб і для інших CPUID
```

Проте, зберігання четвірок 32-бітових значень можна запрограмувати, як здається, і по-іншому – на розсуд програміста.

Вивід інформації

Текстова інформація може відображатися достатньо просто викликом стандартних діалогових вікон MessageBox. Для цього достатньо сформувати у двох буферах потрібний текст: основний і текст заголовку:

```
.data
  Text db 256 dup(0)
  Caption db 32 dup(0)

.code
  . . .          ;формування вмісту буферів Text, Caption
  invoke MessageBoxA, 0, ADDR Text, ADDR Caption, 0
```

Як сформувати вміст буферів тексту? Для даної роботи потрібно виводити значення, отримані командами CPUID. Числові значення можна показувати у шістнадцятковому коді. Для цього потрібно перетворювати 32-бітові значення типу DWORD у 8 символів шістнадцяткових цифр. Таке перетворення можна виконати за допомогою окремої процедури **DwordToStrHex**, яка пропонується нижче

```
; ця процедура записує 8 символів HEX коду числа
; перший параметр - 32-бітове число
; другий параметр - адреса буфера тексту
DwordToStrHex proc
   push ebp
   mov ebp,esp
   mov ebx,[ebp+8]      ;другий параметр
   mov edx,[ebp+12]     ;перший параметр
   xor eax,eax
   mov edi,7
@next:
   mov al,dl
   and al,0Fh           ;виділяємо одну шістнадцяткову цифру
   add ax,48            ;так можна тільки для цифр 0-9
   cmp ax,58
   jl @store
   add ax,7             ;для цифр A,B,C,D,E,F
@store:
   mov [ebx+edi],al
   shr edx,4
   dec edi
   cmp edi,0
   jge @next
   pop ebp
   ret 8
DwordToStrHex endp
```

Процедура **DwordToStrHex** викликається наступним чином:

```
push ...                ;числове 32-бітове значення
push offset ...         ;адреса буферу тексту
call DwordToStrHex
```

Формування символів шістнадцяткових кодів для значень EAX, EBX, ECX, EDX, збережених у масиві **res**, і запис у буфер тексту може бути запрограмоване, наприклад, так:

```
.data
  res dd 256 dup(0)
  Text db 'EAX=xxxxxxxx',13,10,
          'EBX=xxxxxxxx',13,10,
          'ECX=xxxxxxxx',13,10,
          'EDX=xxxxxxxx',0
  Caption db "Результат CPUID 0",0


.code
  . . .                       ;інший програмний код (заповнення масиву res)

  push [res]                  ;значення регістру EAX з масиву res
  push offset [Text+4]        ;адреса, куди записуються 8 символів
  call DwordToStrHex
  push [res+4]                ;значення регістру EBX з масиву res
  push offset [Text+18]
  call DwordToStrHex
  push [res+8]                ;значення регістру ECX з масиву res
  push offset [Text+32]
  call DwordToStrHex
  push [res+12]               ;значення регістру EDX з масиву res
  push offset [Text+46]
  call DwordToStrHex
  invoke MessageBoxA, 0, ADDR Text, ADDR Caption, 0
```

У результаті можна отримати, наприклад, такий текст:

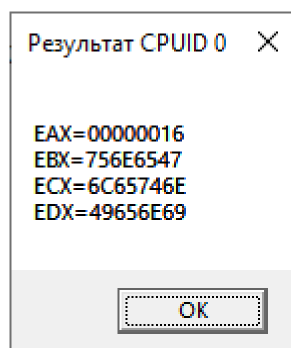


Рис. 1.6. Відображення значень регістрів після виконання команди CPUID

Як окремий випадок можна відзначити вивід імені процесора, яке записується у регістри EBX, ECX та EDX після виконання команди CPUID з параметром 0. Особливість полягає у тому, що у вказані регістри тут записуються безпосередньо однобайтові ASCII коди символів – по чотири символи в регістр. Тоді ці коди символів можна прямо вставити у відповідні байти текстового буферу. Наприклад:

```
.data
  Vendor db 16 dup(0)
  CaptionVendor db "CPUID 0 Vendor string",0
  . . .                    ;інші дані
.code
  mov eax, 0
  cpuid
  mov dword ptr[Vendor], ebx
  mov dword ptr[Vendor+4], edx
  mov dword ptr[Vendor+8], ecx
  . . .                    ;інший програмний код
  invoke MessageBoxA, 0, ADDR Vendor, ADDR CaptionVendor, 0
```

У діалоговому вікні буде відображатися наступне:

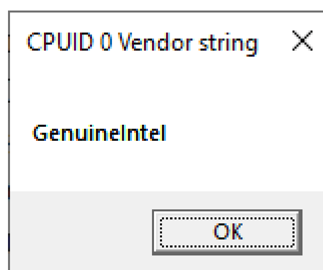


Рис. 1.7. Один з результатів виконання команди CPUID – ім’я процесора

Засоби розробки програм на асемблері

Для створення програми, написаної на асемблері, загалом, потрібні: редактор вихідного тексту, компілятор, лінкер. Вихідний текст на асемблері записується у файл *.asm, який потім компілюється, лінкується і створюється виконуваний файл *.exe. Наприклад, файл вихідного тексту **myprog.asm** компілюється у об’єктний файл **myprog.obj**. Потім на основі об’єктного файлу myprog.obj лінкер записує виконуваний файл **myprog.exe**. Виконуваний файл є результатом роботи програміста – користувача пакету розробки програм.

Окрім вказаних засобів, корисно мати налагоджувач (debugger) для відстеження роботи програми та контролю помилок. У пакеті MASM32 його немає.

Набагато потужнішим є середовище розробки програмного забезпечення **Microsoft Visual Studio**. У цьому середовищі можна створювати проекти з використанням декількох мов програмування, наприклад, C++, C#, асемблер та інших. Незважаючи на те, що **Microsoft Visual Studio** орієнтоване, у першу чергу, на мови програмування високого рівня, але це середовище надає широкі можливості використовувати асемблер і створювати різноманітні проекти як чисто на асемблері, так і комбінуванням модулів на асемблері з програмним кодом на мовах високого рівня, зокрема, C++.

Методичні рекомендації

Рекомендується спочатку інсталювати пакет **MASM32** та створити проект найпростішої програми, яка виводить на екрані типове повідомлення-вітання. Після цього інсталювати **Microsoft Visual Studio** і створити проект у його середовищі. Чому саме у такій послідовності? По-перше, MASM32 є набагато простішим для початківця. По-друге, при програмуванні на асемблері у середовищі Microsoft Visual Studio можуть знадобитися деякі файли зі складу пакету MASM32, зокрема заголовкові файли *.inc для функцій API Win32.

Робота з MASM32

Інсталяція пакету MASM32 та підготовка до роботи

Спочатку необхідно інсталювати пакет MASM32. Цей пакет являє собою програмне забезпечення, яке є вільним (freeware) для розповсюдження та використання, у тому числі для навчальних цілей. Знайти інсталяційні файли можна у мережі Інтернет за адресою http://www.masm32.com. Після завантаження потрібних файлів перевірте їх на відсутність комп'ютерних вірусів. Потім запустіть на виконання файл **install.exe**. Дочекайтеся повідомлення про успішне завершення інсталяції MASM32.

Після завершення інсталяції на жорсткому диску комп'ютера буде записано множину різноманітних файлів – у тому числі виконувані файли програм. Увага! Перед тим, як щось робити з отриманими файлами, обов'язково знову перевірте усі файли на відсутність комп'ютерних вірусів.

Порада: задля безпеки вилучіть з жорсткого диску у папках **\masm32\examples** усі файли ***.exe** скомпільованих прикладів (попередньо записавши їхні імена – ці імена буде корисно знати для подальшого вивчення пакету MASM32).

Створення проекту у середовищі MASM32

У розділі (локальному диску), де міститься інстальований MASM32, створіть робочу папку вашої програми. Наприклад, якщо MASM32 розташовується у E:\masm32, то створіть робочу папку, наприклад, E:\work\lab1.

Потім викличте програму MASM32 Quick Editor (виконуваний файл \masm32\qeditor.exe).

Введіть текст найпростішої програми для Windows (рис. 8). Потім запишіть цей текст у файл типу *.asm у робочу папку на локальному диску – наприклад, як E:\work\lab\lab1.asm. Зробити це можна через меню "File - Save As".

У такий спосіб створюються файли вихідного тексту. Дана програма містить лише один файл вихідного тексту. Складніші програми можуть містити декілька файлів у робочій папці проекту.

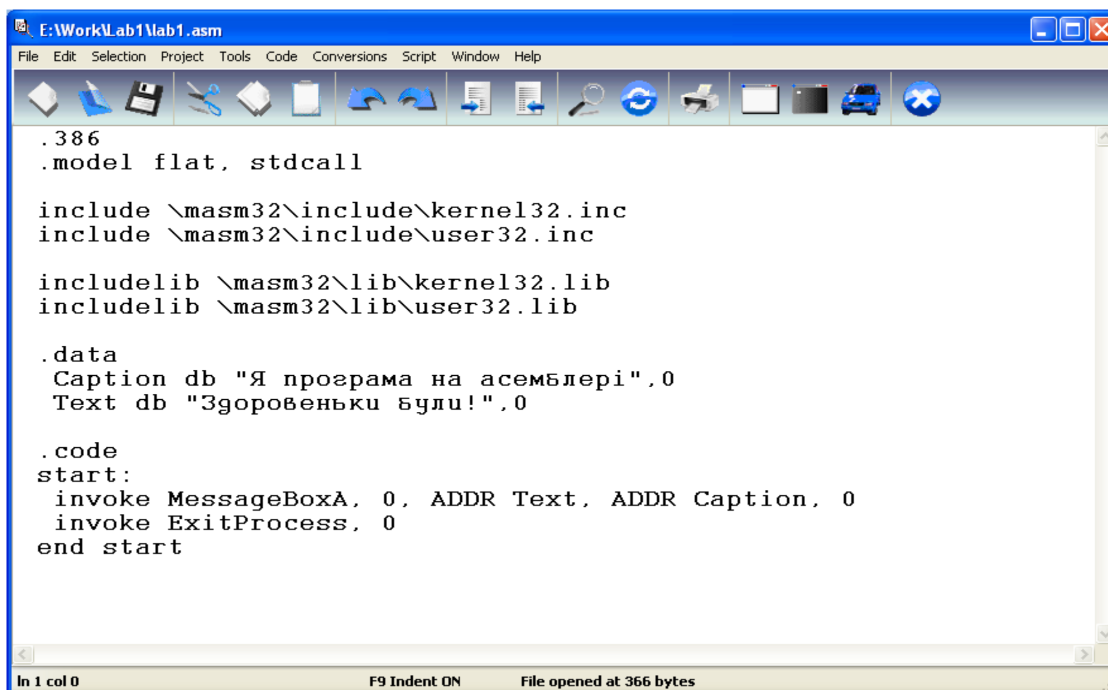


Рис. 1.8. Вихідний текст у вікні MASM32 Quick Editor

Компіляція та створення виконуваного файлу програми

Виберіть меню "Project - Assemble ASM File". Якщо компілятор не знайде помилок, то у робочу папку буде записаний файл lab1.obj.

Наступним кроком буде лінкування. Виберіть меню "Project – Link OBJ File". У разі відсутності помилок лінкер запише виконуваний файл програми lab1.exe.

Необхідно вказати, що компіляцію та лінкування можна об'єднати – виконати за один раз вибором меню "Project – Assemble & Link". Проте і у цьому випадку послідовність обробки вихідного тексту буде тою самою – спочатку компіляція, потім лінкування. Інформація про виконання цієї роботи буде відображатися у вікні командного рядка (рис. 1.9)

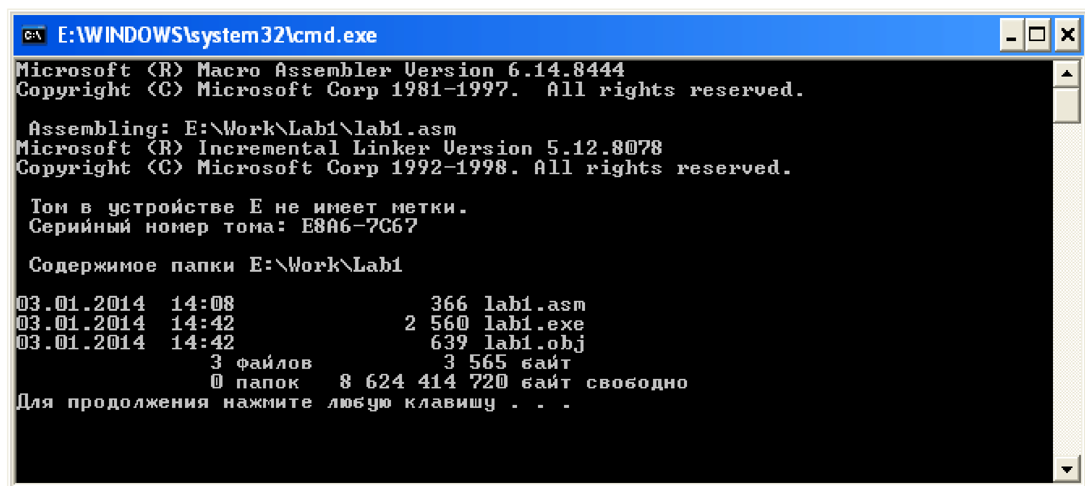


Рис. 1.9. Інформація про виконання компіляції та лінкування

Перевірка роботи програми

Таким чином, у робочій папці з'явився виконуваний файл lab1.exe. Щоб перевірити, як працює новостворена програма lab1.exe, її можна викликати засобами операційної системи, або безпосередньо у середовищі MASM32 Editor. Для цього виберіть меню "Project – Run Program". Наша перша програма на асемблері виводить текст вітання у діалоговому вікні

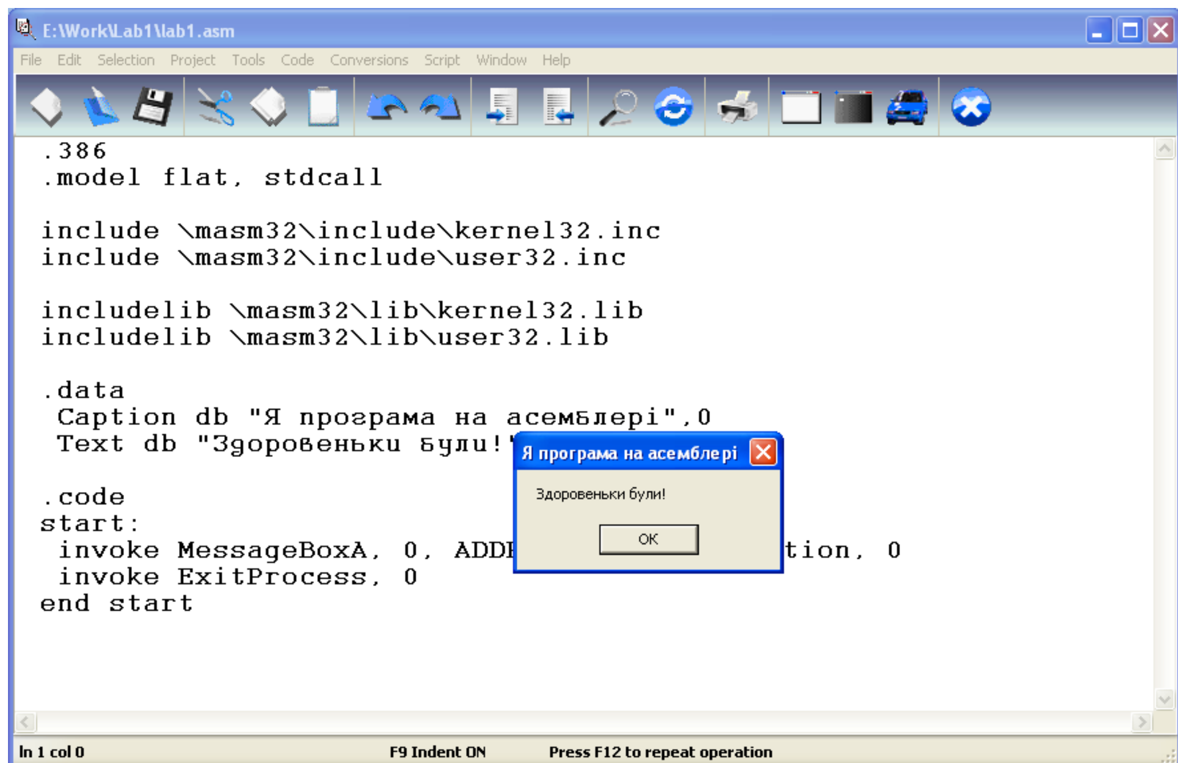


Рис. 1.10. Виклик програми на виконання

Редагування вихідного тексту програми

Потрібно змінити, відредагувати вихідний текст програми згідно варіанту завдання. Це виконується у програмі MASM32 Quick Editor.

Для вирішення завдання необхідно відповідно оформити потрібний текст директивою DB у секції .DATA. Студент повинен самостійно розібратися у питанні, як можна записувати текст з декількох рядків у діалоговому вікні. Порада: для цього слід використовувати пари символів з кодами 13 та 10 у якості роздільників рядків.

Робота з Microsoft Visual Studio

Інсталяція Microsoft Visual Studio та підготовка до роботи

Знайти інсталяційні файли можна на офіційному сайті корпорації Microsoft. Зокрема, для навчальних цілей рекомендується безкоштовна версія Microsoft Visual Studio Community від 2019 та пізніше. Її сучасні релізи можна скачати з сайту Microsoft https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes, а потім зареєструвати.

Створення проекту у середовищі MS Visual Studio

При створенні проекту чисто на асемблері потрібно враховувати особливості середовища – для проектів MS Visual Studio у переліку мов програмування асемблер відсутній. Рекомендація: створювати проект С++. У стартовій панелі Visual Studio виберіть “Create a new project”.

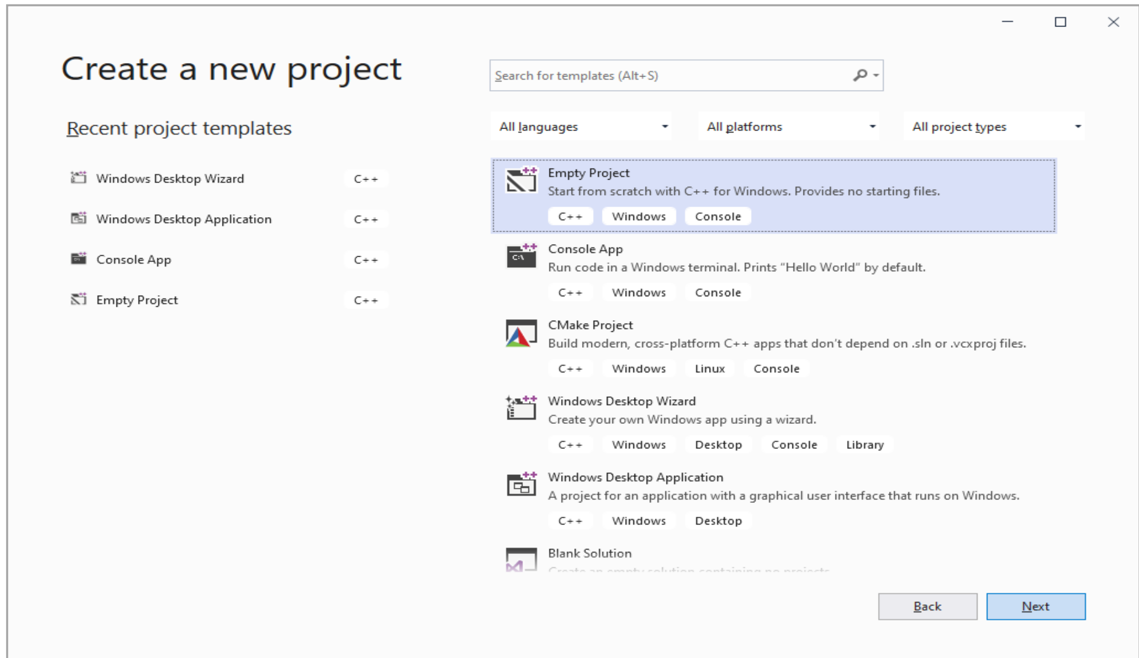


Рис. 1.11. Початок створення проекту Visual Studio

Треба обрати порожній проект (Empty Project). Далі у наступному вікні потрібно ввести ім’я нового проекту та його розташування на диску. Нехай ім'я програми та ім'я рішення буде, наприклад, MyProg. Окрім цього, розташуємо проект у окрему папку, наприклад, D:\Work\SP\

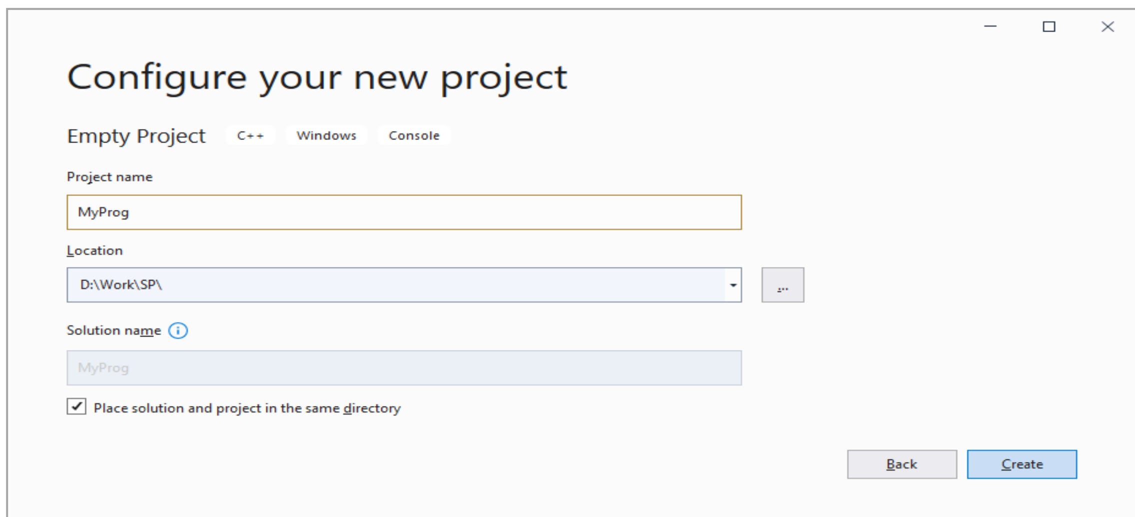


Рис. 1.12. Ім’я нового проекту та його розташування

Після натискування кнопки Create створиться новий порожній проект і Visual Studio перейде у стан очікування подальших дій програміста.

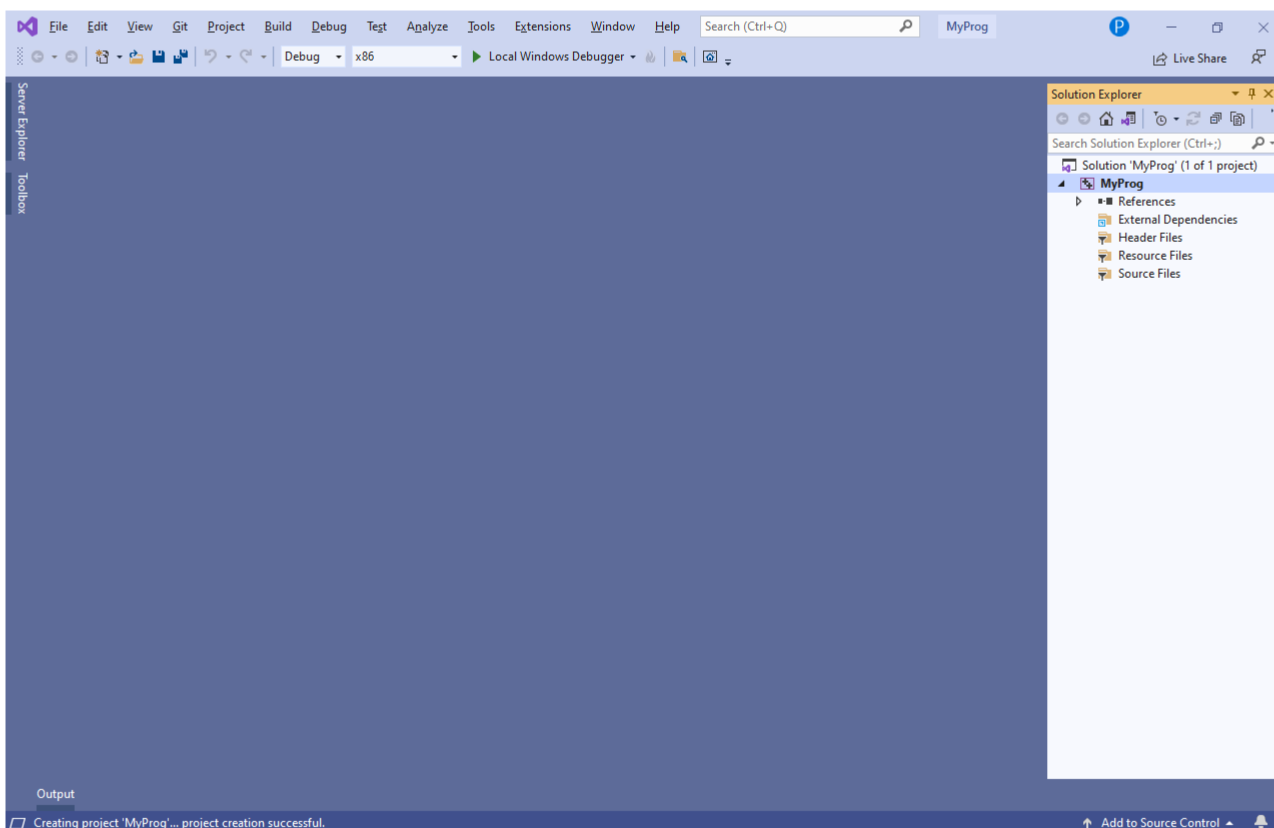


Рис. 1.13. Вигляд середовища розробки програми

У вікні “Solution Explorer” (праворуч) відображаються розділи програмного коду та вихідних текстів. Оскільки ми створили порожній проект, то у цих розділах немає жодних вихідних текстів

Потрібно додавати тексти програми у папку **Source Files** та у інші папки (якщо потрібно). Проте, спочатку необхідно дещо налагодити робоче середовище Visual Studio, інакше асемблерні тексти не будуть сприйматися.

Налагодження підтримки мови асемблера

У вікні "Solution Explorer" встановіть курсор на імені проекту і клацніть правою кнопкою миші. У спливаючому меню виберіть пункт "Build Dependencies" і далі "Build Customization". Має з'явитися діалогове вікно Visual C++ Build Customization Files (рис. 1.14)

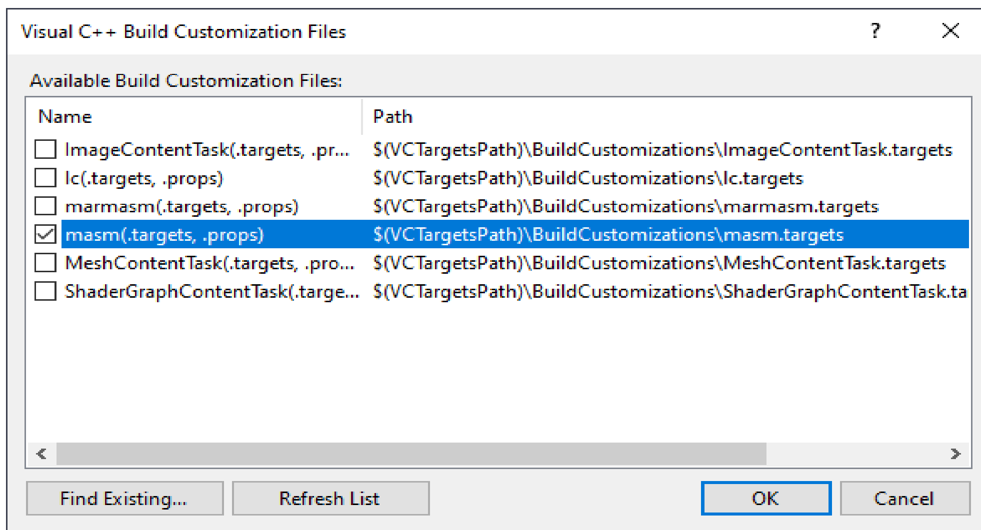


Рис. 1.14. Опція підтримки асемблера у Visual Studio

Виберіть пункт "masm", та зробіть позначку у квадратику. Натисніть ОК. Таким чином, у проекті активується вбудований у Visual Studio асемблер MASM, який буде компілювати файли вихідних текстів *.asm.

Додавання вихідних текстів у проект

Нам потрібно додати у проект файл вихідного тексту на асемблері. Будемо вважати, що тексту поки що немає і ми плануємо його писати безпосередньо у середовищі Visual Studio. Вкажіть курсором розділ "Source Files", клацніть правою кнопкою миші, і потім - "Add" – "New Item…"

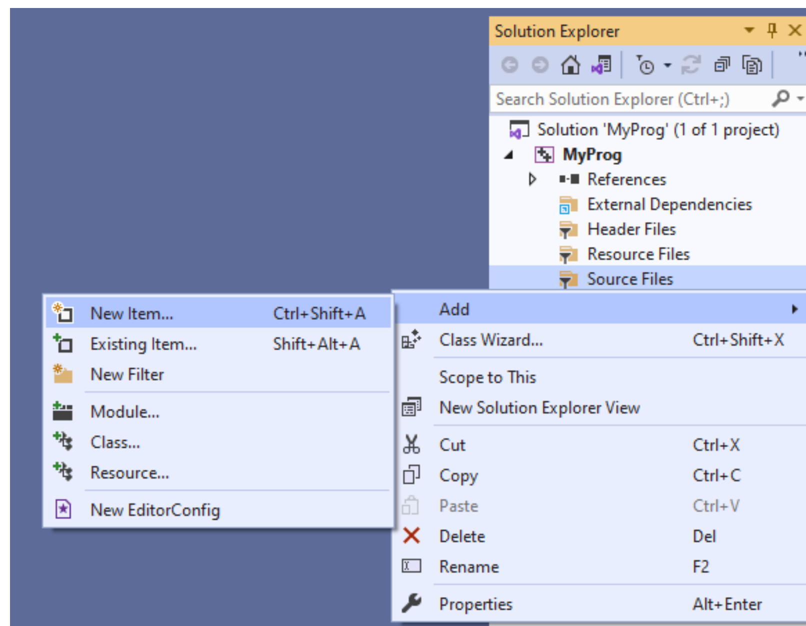


Рис. 1.15. Додавання файлів у проект

Далі з'являється вікно, у якому буде запропоновано вибрати тип файлу та його назву. Нам потрібен файл вихідного тексту Асемблера – проте у цьому списку таких типів файлів немає. Можемо вибрати майже будь-який тип, наприклад, тип файлів .cpp, але ввести ім'я та розширення – MyProg.asm.

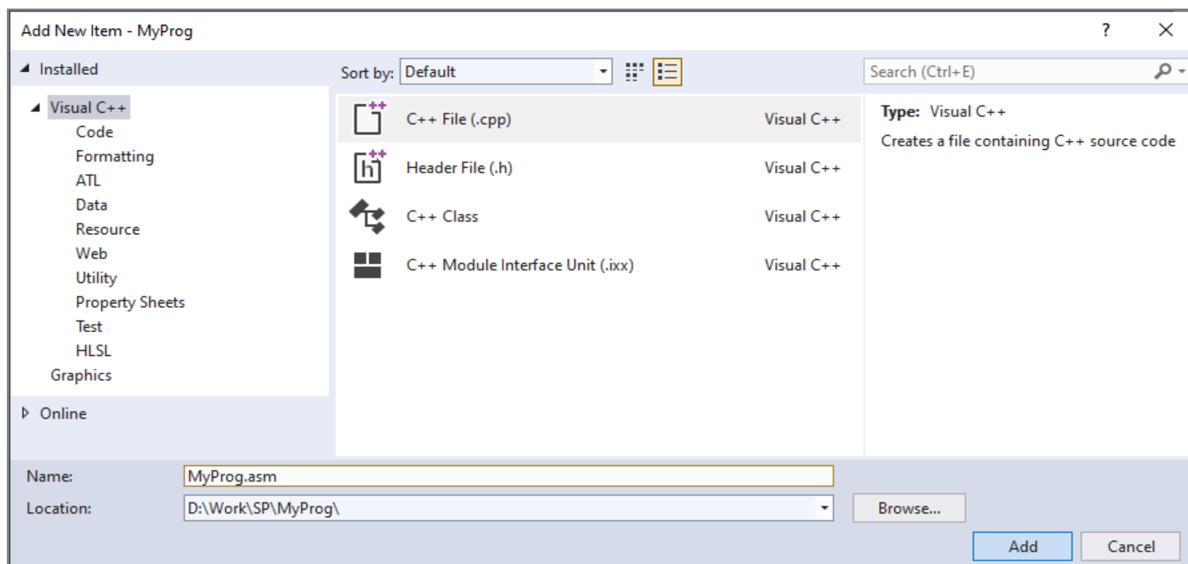


Рис. 1.16. Визначення типу та імені файлу вихідного тексту

У результаті у вікні “Solution Explorer” у розділі Source Files має з'явитися MyProg.asm. Також у середовищі Visual Studio відкривається нове порожнє вікно з ім'ям MyProg.asm, у якому ми будемо писати текст програми. Наприклад, такий:

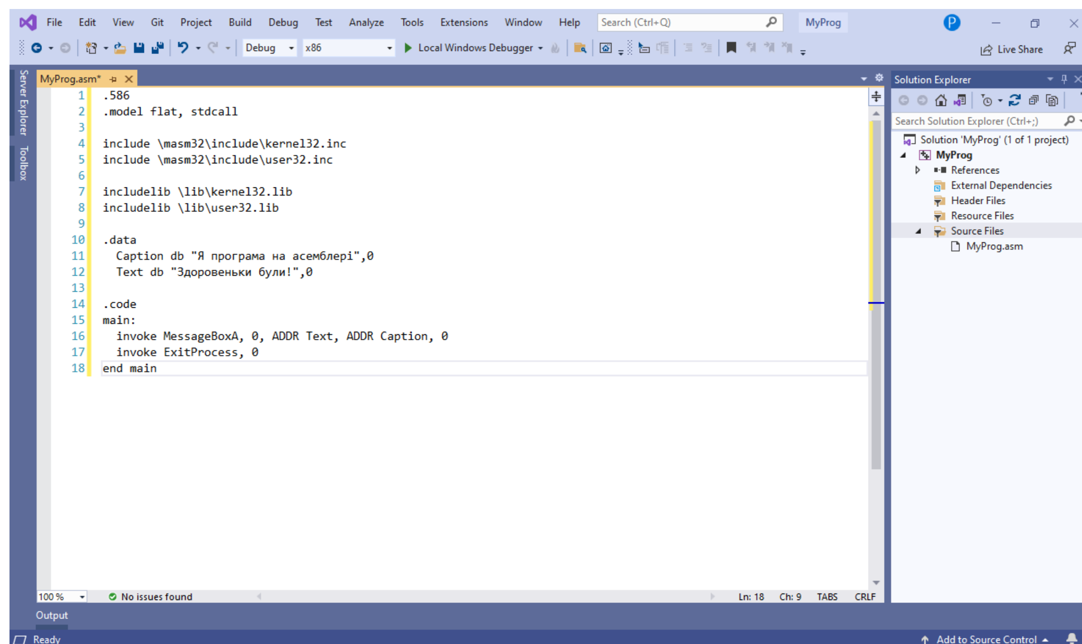


Рис. 1.17. Текст найпростішої програми на асемблері у вікні Visual Studio

Підключення бібліотек

Для роботи з проєктом потрібно ще вирішити питання щодо підключення бібліотек. Програма викликає дві функції Windows, які доступні через інтерфейс API Win32:

- **MessageBoxA** для показу простого діалогового вікна. Ця функція міститься у системній DLL – **user32.dll**. Для побудови проекту лінкер використовує бібліотеку імпорту **user32.lib**.
- **ExitProcess** – цю функцію повинна викликати будь-яка програма для коректного завершення. Вона викликається з системної DLL – **kernel32.dll**. Для побудови проекту використовується бібліотека імпорту **kernel32.lib**.

Підключення до проекту вказаних вище бібліотек описується рядками

```
includelib \lib\kernel32.lib
includelib \lib\user32.lib
```

Для компіляції викликів функцій **MessageBoxA** та **ExitProcess** потрібно знати, які вони мають аргументи. Опис аргументів міститься у заголовочних файлах ***.inc**, які включені у текст директивами include:

```
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
```

Де знайти файли **kernel32.inc** та **user32.inc**? Серед файлів Visual Studio їх немає. Вказані файли можна взяти з комплекту MASM32 (дивіться першу частину лабораторної роботи). Потрібно створити на диску, на якому записується поточний проект, наприклад, диску D, папку **d:\masm32\include** і записати туди вказані вище файли.

Компіляція та створення виконуваного файлу програми

Для створення виконуваного файлу програми потрібно спочатку скомпілювати вихідний текст, а потім лінкер запише файл – MyProg.exe. Потім викликати програму на виконання. Усе це можна зробити одразу через меню "Debug – Start Debugging".

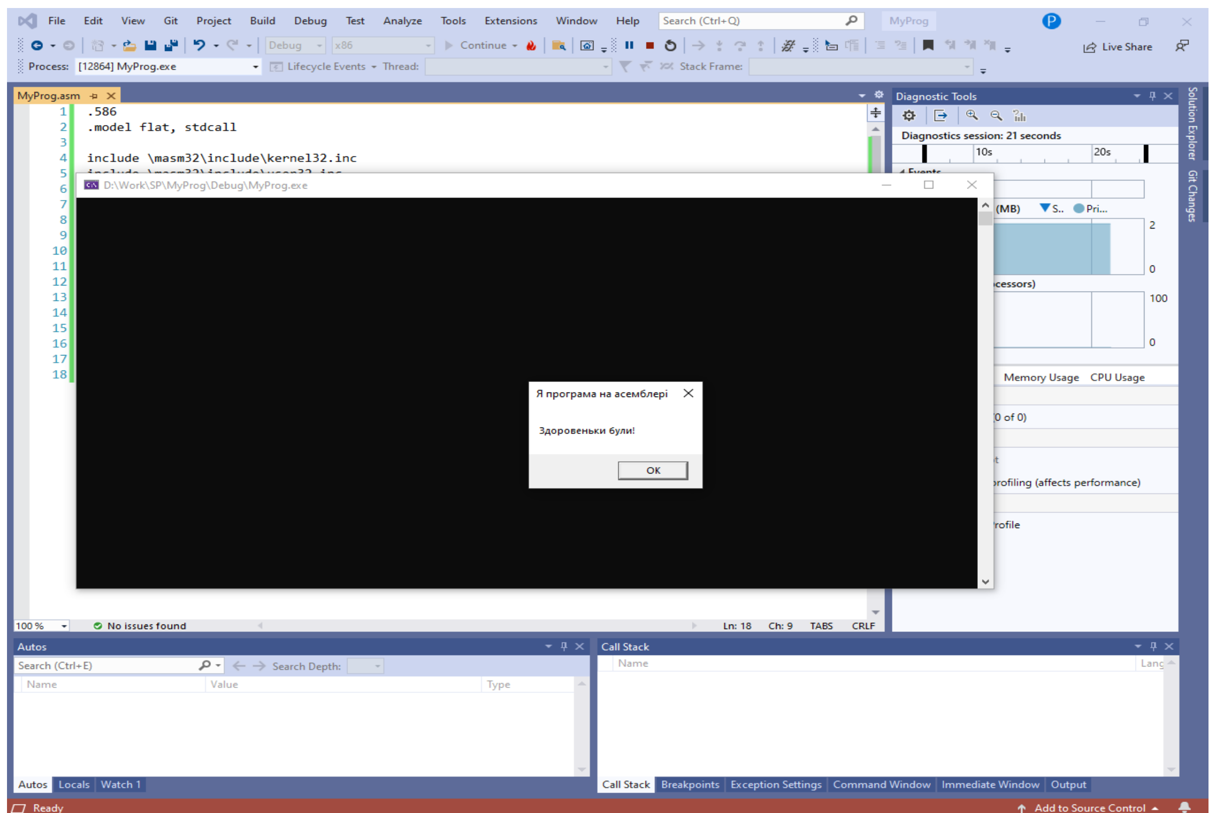


Рис. 1.18. Вигляд вікна програми MyProg на тлі середовища MS Visual Studio

Натисніть кнопку "OK" – програма закінчить свою роботу. А також, потрібно закрити вікно консолі. Це якось незручно. І, взагалі, навіщо нам консоль?

Виберіть меню властивостей проекту (рис. 2.19)

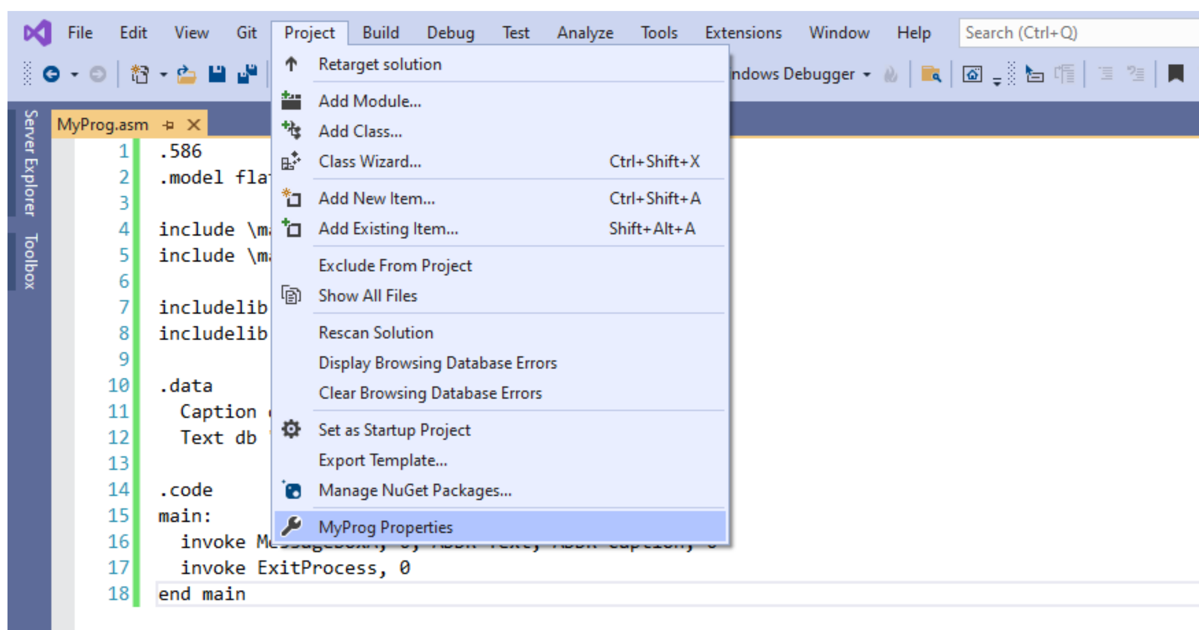


Рис. 1.19. Доступ до опцій властивостей проекту

У діалоговому вікні властивостей проекту треба знайти потрібну опцію

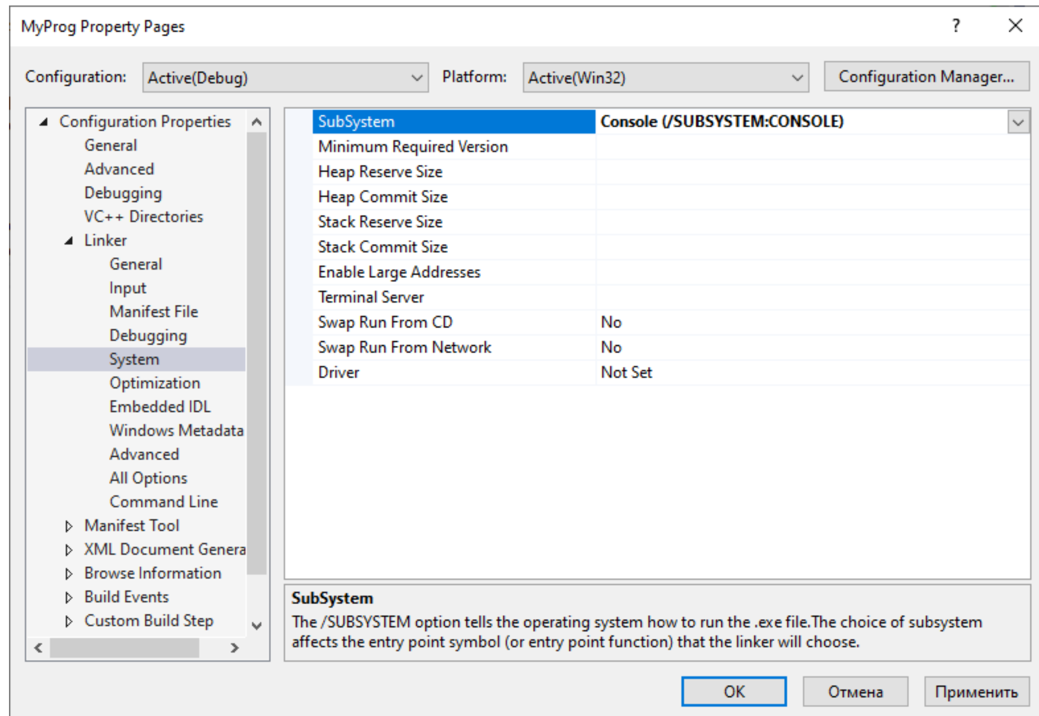


Рис. 1.20. Вікно властивостей проекту Visual Studio

Як прибрати вікно консолі? Для цього треба вибрати підсистему (subsystem) підтримки виконання. Замість Console виберіть Windows

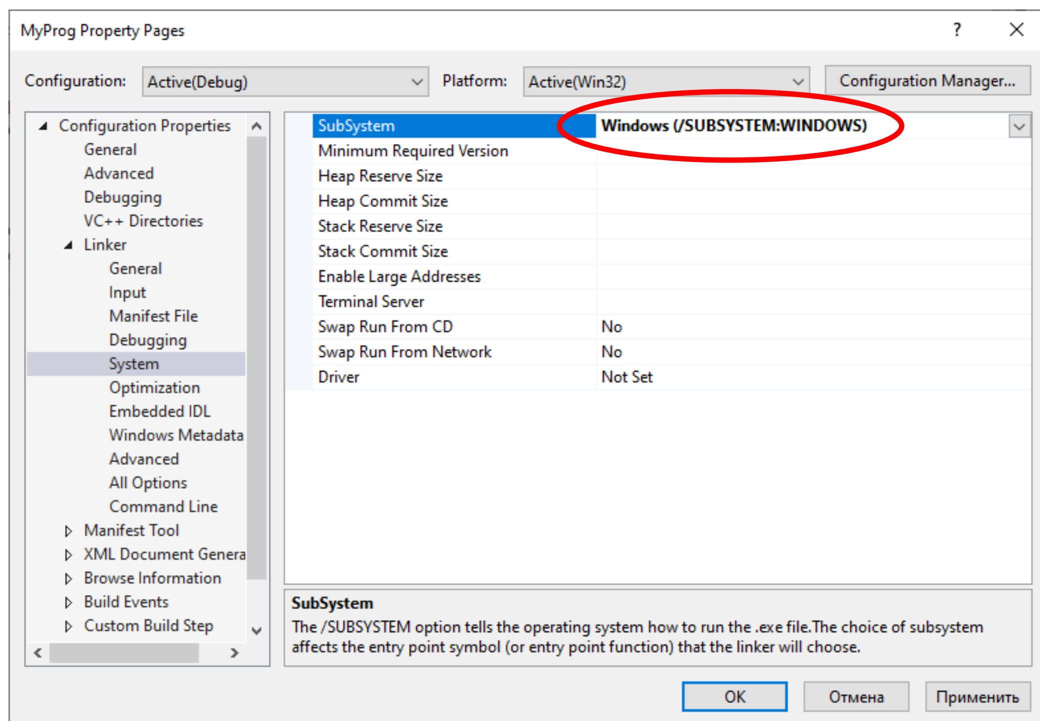


Рис. 1.21. Зміна підсистеми виконання програм

Знову через меню “Start Debugging” перевіримо роботу програми – зайвого вікна консолі вже не повинно бути (рис. 1.22)

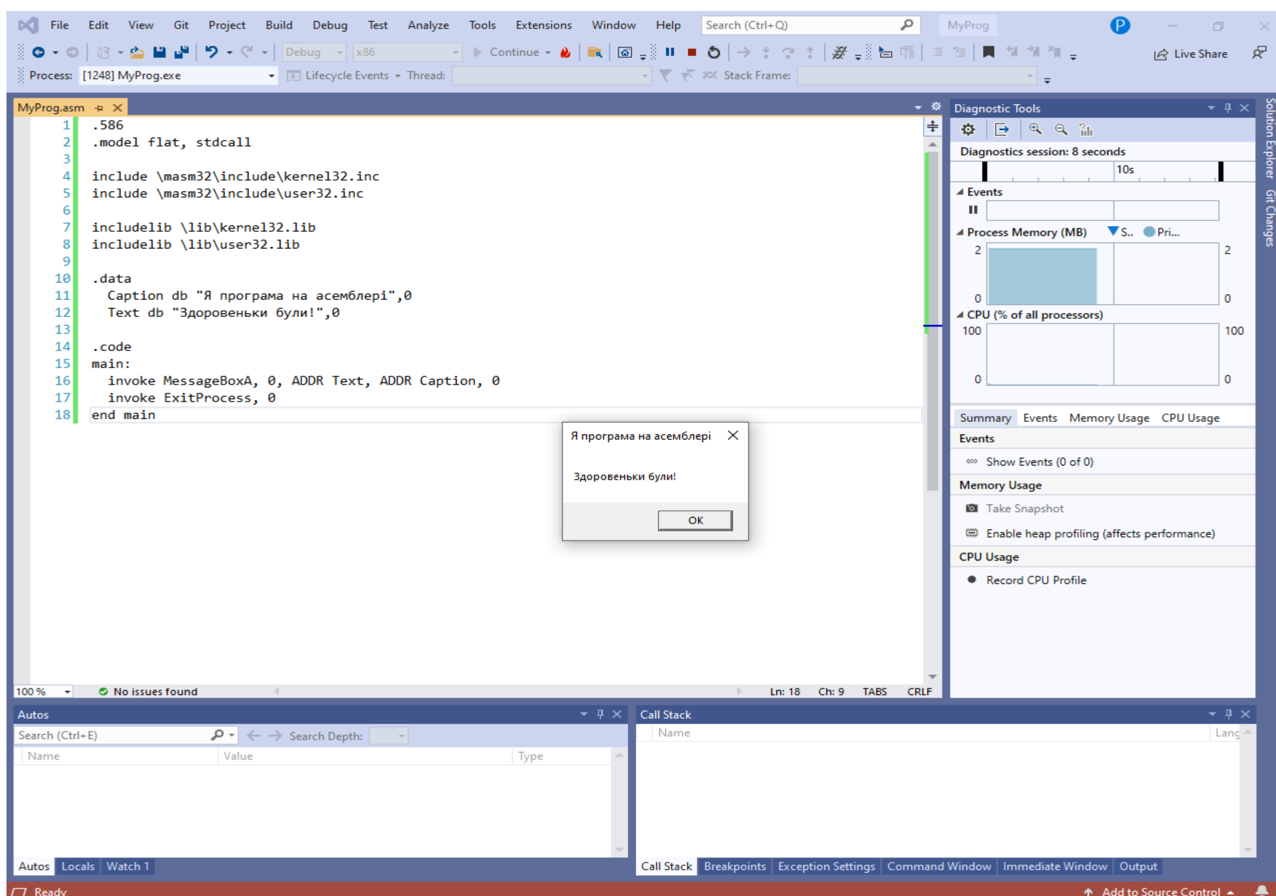


Рис. 1.22. Найпростіша програма на асемблері у середовищі Visual Studio

Debug- та Release-версії програми

За умовчанням при створенні нового проекту середовище Visual Studio відпрацьовує Debug-версію нашої програми. Що це означає? У машинний код вставляються команди, які дозволяють контролювати роботу програми, налагоджувати її. У робочій папці створюється вкладена папка Debug – у нашому проекті це \MyProg\Debug у яку після кожного успішного сеансу лінкування записується файл MyProg.exe. Debug-версія програми є деяким тимчасовим зразком, який має сенс тільки на етапі розробки та налагодження.

Якщо програміст хоче розповсюджувати власну програму, то Debug-версія для цього не дуже придатна з багатьох причин.

Фінальна версія exe-файлу програми, яка призначена для розповсюдження і використання на інших комп'ютерах, зветься Release-

версією. Щоб налаштувати конфігурацію середовища Visual Studio на створення цієї версії, виберіть виберіть "Release" у списку нижче меню

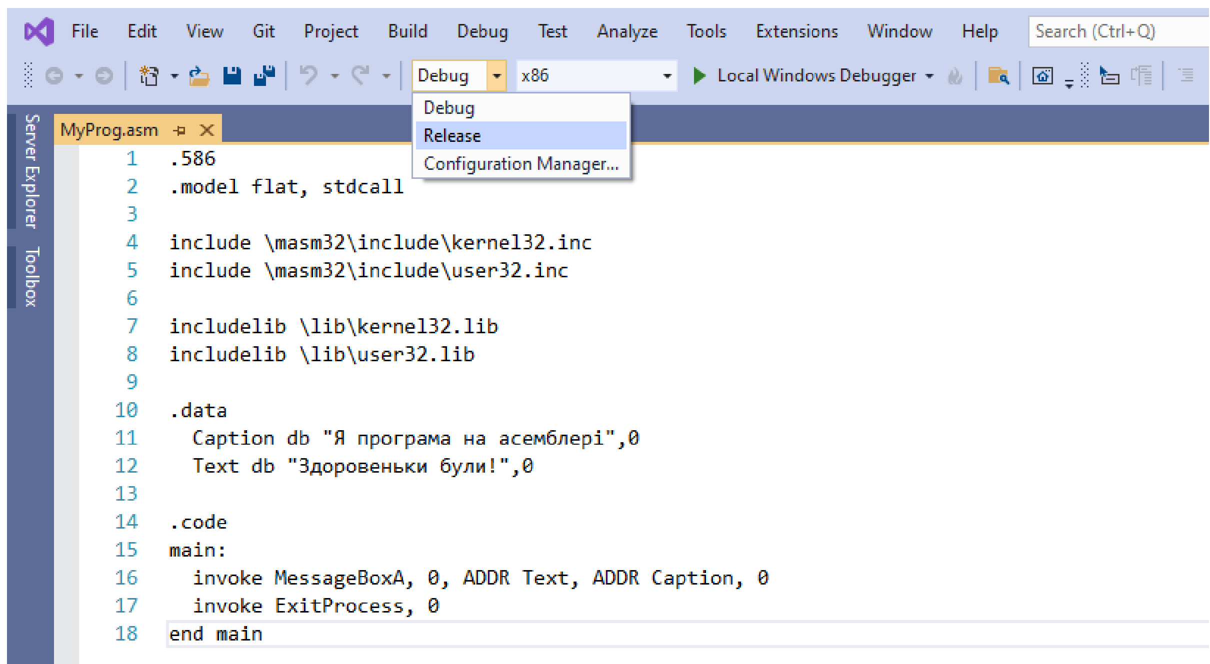


Рис. 1.23. Зміна конфігурації Debug на Release

Середовище починає працювати трохи по-іншому. Виконуваний файл буде записуватися у вкладену папку Release - у нашому проекті це буде \Myprog\Release.

Необхідно враховувати відмінність багатьох параметрів роботи середовища для різних конфігурацій. Це стосується також і параметрів (опцій) роботи компілятора та лінкера. Може так трапитися, що компілятор або лінкер можуть видати повідомлення про помилку, якої не було у Debug-конфігурації.

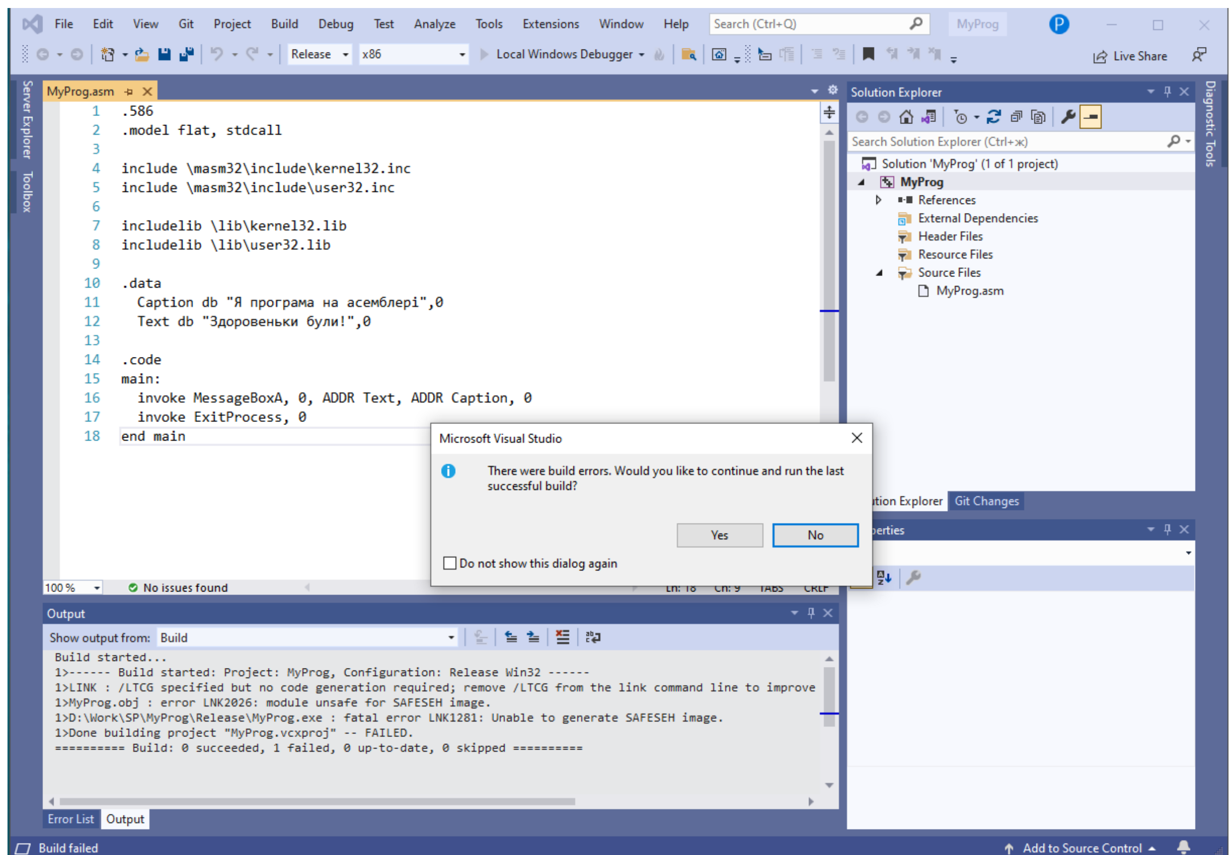


Рис. 1.24. Лінкер зупинив побудову і повідомив про помилки

Зазвичай, компілятор та лінкер надають деякі відомості про помилки, зокрема, можна придивитися до вмісту вікна "Output" (рис. 1.25)

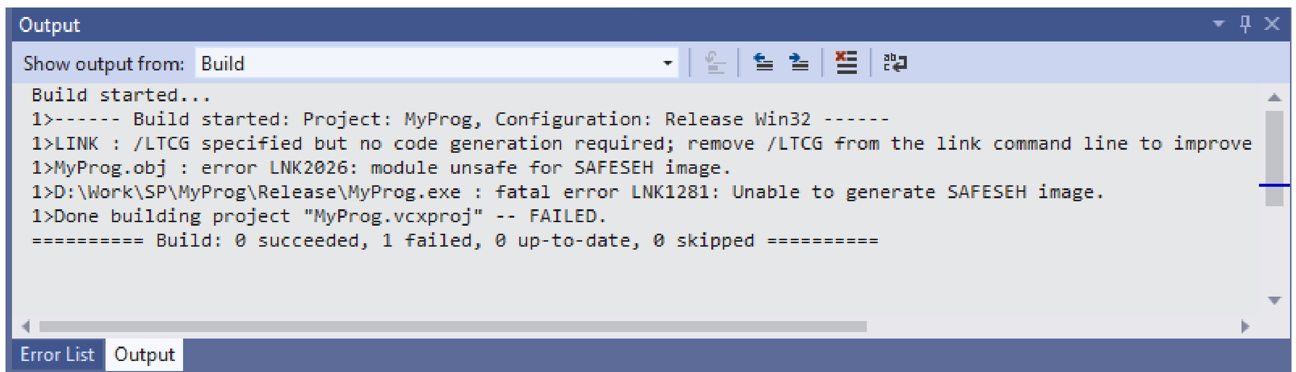


Рис. 1.25. Вікно Output з коментарями щодо помилок

У повідомленнях мова йде про поки що незнайомі нам /LTCG та SAFESEH. У будь-якому випадку наявності помилок потрібно у вікні повідомлень натиснути кнопку "No" і шукати вирішення проблеми. Середовище призупиняє роботу і надає список помилок (рис. 1.26).

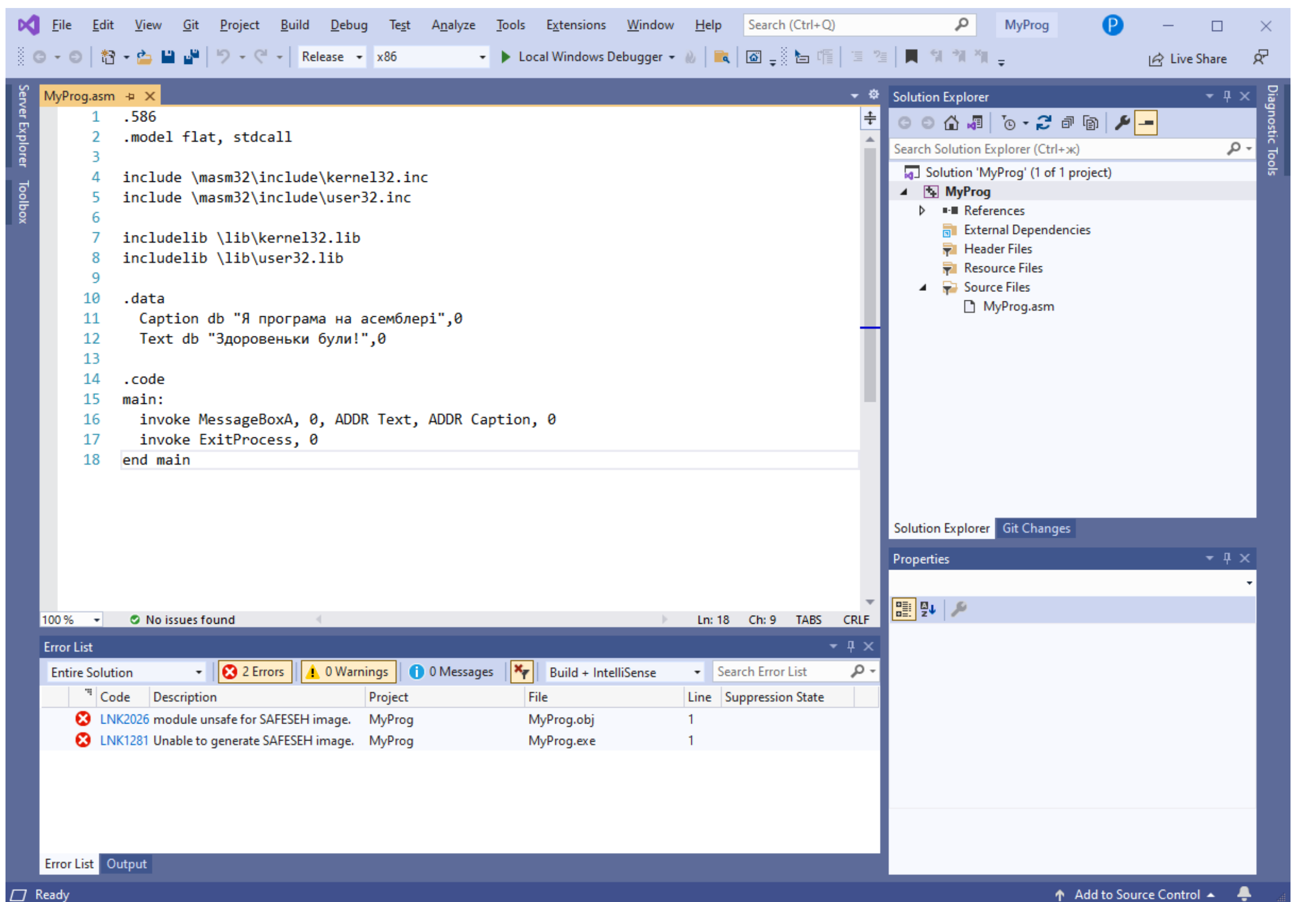


Рис. 1.26. Список помилок у вікні "Error List" знизу

Де шукати якійсь SAFESEH? Оскільки таку помилку видає лінкер, то можна спробувати пошукати у налаштуваннях лінкера. Вибираємо пункт меню Project – Properties і у вікні властивостей проекту розкриваємо розділ Linker. На сторінці Advanced міститься рядок із SAFESEH. Вказуємо No замість Yes. Після внесення змін, натискуємо кнопку OK (рис. 1.27).

Примітка. Опція SAFESEH:NO може призвести до створення менш захищеного машинного коду, який буде записано у виконуваний exe-файл. Про особливості машинного коду та вмісту виконуваних файлів з урахуванням механізму SEH інформацію можна прочитати у бібліотеці MSDN.

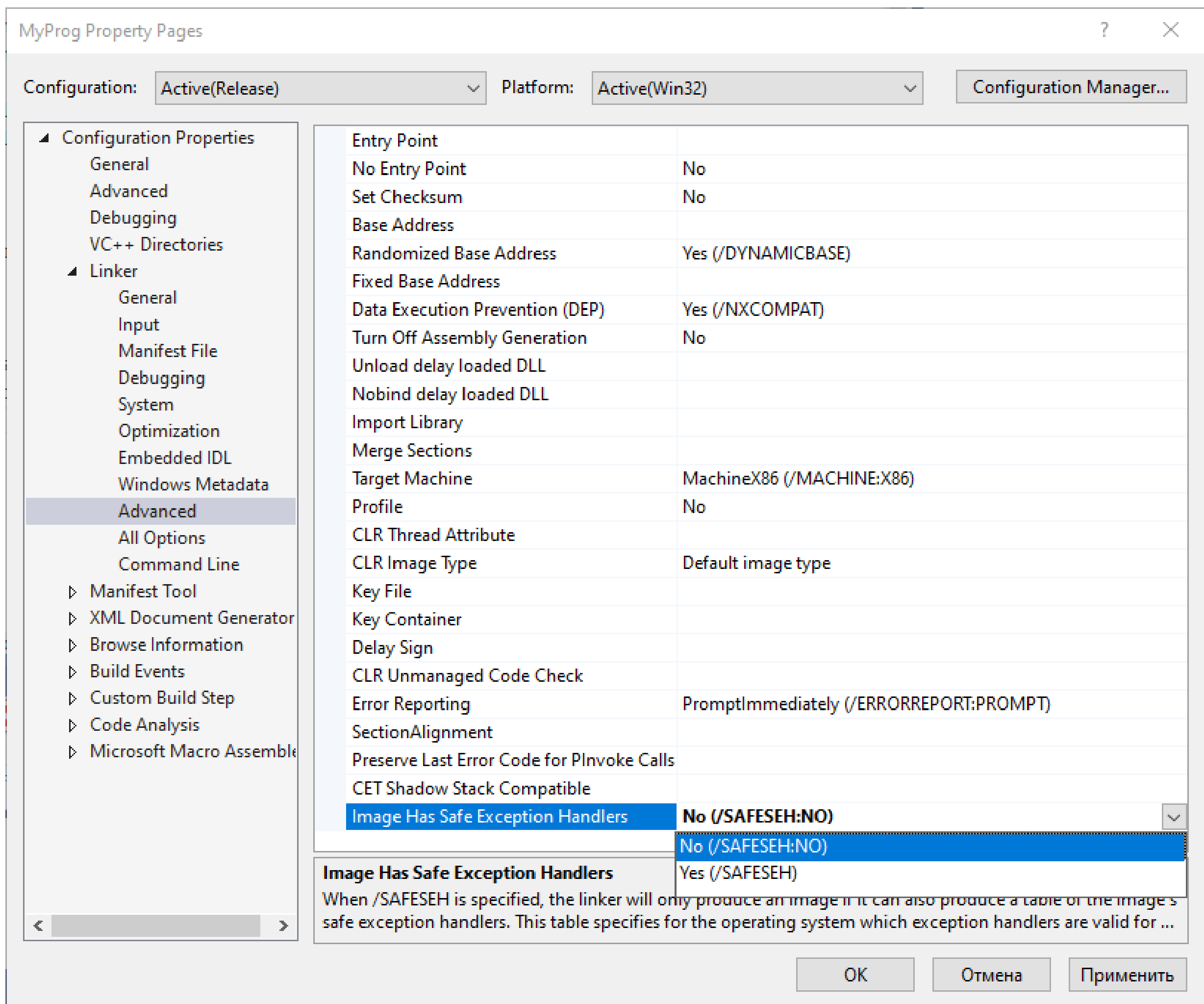


Рис. 1.27. Одна зі сторінок опцій налаштування лінкера MS Visual Studio

Можна вказати ще на один аспект при використанні мови програмування Асемблер у середовищі Visual Studio – точка входу у програму. У тексті нашої програми вона позначена міткою "**main:**". Зазвичай Visual Studio успішно автоматично знаходить точку входу в програму, навіть якщо вона позначається не "**main:**" а будь-якою іншою міткою, наприклад, "**start:**" або взагалі довільним набором англомовних символів. Проте, інколи можуть виникати помилки, пов'язані з тим, що точка входу автоматично не знаходиться. Тоді її треба для Visual Studio вказати явно (за умовчанням не вказується). Для цього через меню меню Project – Properties у вікні властивостей проекту розкриваємо розділ Linker. У першому рядку знаходимо Entry Point і вписуємо туди потрібну назву – **main**, або те що треба (рис. 1.28)

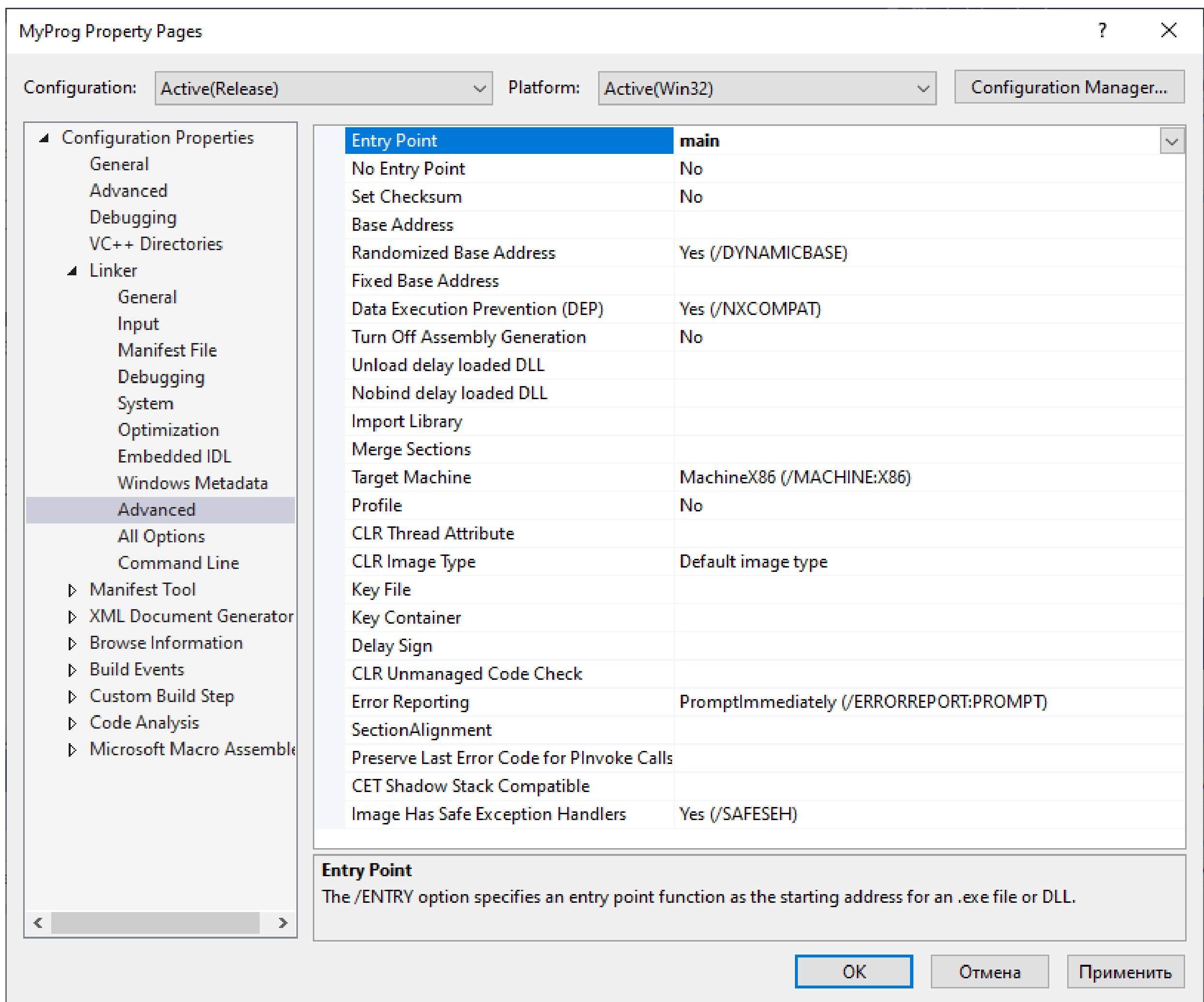


Рис.1.28. Вказування імені точки входу

Налагодження програми, огляд та аналіз машинного коду

Зазвичай програму потрібно налагоджувати, виправляти помилки, вдосконалювати. Після кожного кроку виправлення помилок можна робити перевірку – спробувати побудувати проект. Це робиться через меню Build. Проте, зазвичай зручніше усі дії виконувати через меню Debug – Start Debugging (кнопка F5 на клавіатурі) – автоматично виконуються усі етапи: компіляція, лінкування і при відсутності помилок програма буде викликана для демонстрації роботи або відстеження ходу її виконання.

В принципі, налагоджувати программу можна і у Release, зокрема, відстежувати виконання коду. Проте, набагато більше можливостей для аналізу та контролю є у конфігурації Debug – саме для цього вона і призначена.

Налаштуйте середовище Visual Studio на Debug-конфігурацію. Потім потрібно встановити точки зупинок (*breakpoints англ.*). Точки зупинки ставляться у лівому стовпчику вікна вихідного тексту. Нехай нам потрібно відстежити роботу програми від початку - ставимо точку на першому рядку:

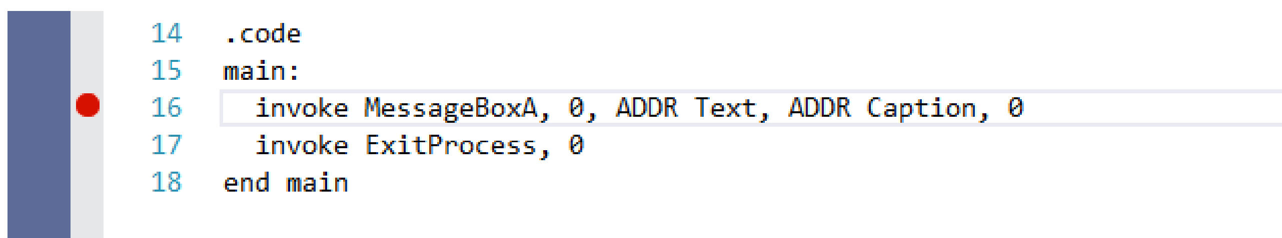


Рис. 1.29. Точка зупинки – червона ліворуч

Розпочинаємо налагодження, натиснувши клавішу F5. Але програма не зупиняється, немов би не помічає точку зупинки. У чому справа? Порада: якщо точка зупинки на першій команді, то розпочинайте налагоджування, натиснувши клавішу F10.

Знову розпочнемо налагодження. Натиснемо клавішу F10. Тепер процес зупиняється у потрібній точці. Для перегляду значень регістрів відкрийте відповідне вікно через меню "Debug – Windows – Registers". Для аналізу створеного компілятором машинного коду відкрийте вікно дизасемблерного коду через меню "Debug – Windows – Disassembly".

У вікні "Disassembly" відображається машинний код, який згенерував компілятор. Це той код, який насправді виконує програма. Проте у вікні дизасемблера надається не власне машинний код (це множина двійкових чисел – кодів команд та даних), а текст у вигляді рядків на мові асемблеру як результат **зворотного перетворення машинного коду у асемблер**. Дизасемблерний текст найбільш близький до машинного коду – хіба що числові коди замінені символічними іменами команд та процедур. Тому цей текст дуже корисний для аналізу програми.

Не зважаючи на те, що ми пишемо вихідні тексти на асемблері, дизасемблерний текст може суттєво відрізнятися від написаного програмістом вихідного тексту. Наприклад, замість **invoke** насправді буде декілька **push**, а потім **call**. Чим більше програміст використовує високорівневих синтаксичних конструкцій певної версії асемблера, тим більше вихідний текст програми відрізнятиметься від дизасемблерного.

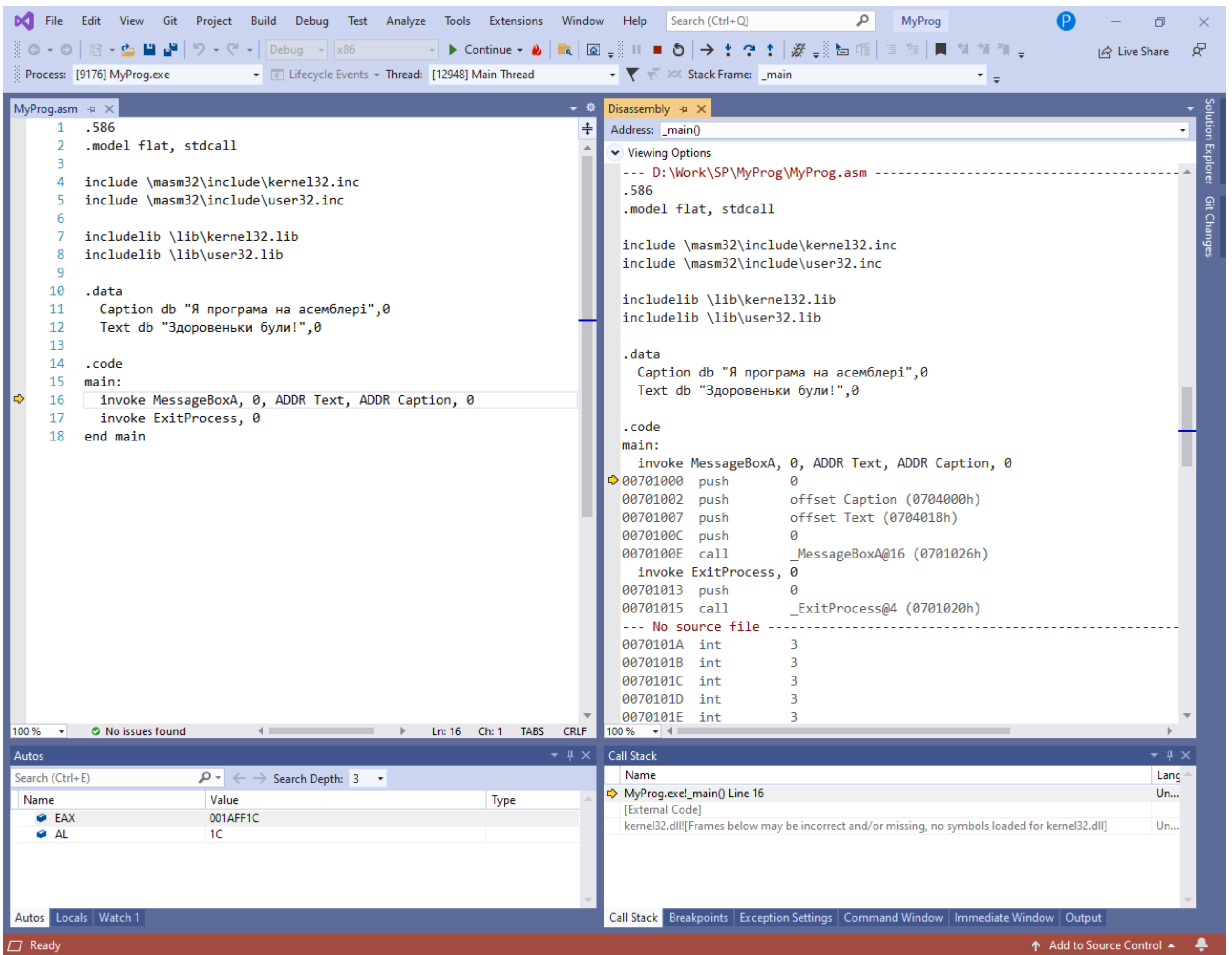


Рис. 1.30. Середовище Visual Studio у режимі стеження за покроковим виконанням програми з показом дизасемблерного коду

Для того, щоб простежити роботу програми по крокам натискуйте клавішу F10 або клавішу F11. Чим вони відрізняються? При натискуванні F11 (Step into) налагоджувач входить у процедури – так можна увійти у процедуру MessageBoxA і простежити її код та процес виконання. При натискуванні F10 (Step over) налагоджувач у середину процедур не входить і спостерігачу здається, що поточна процедура виконується немов би за один крок.

Варіанти завдань та основні вимоги

Кожному студенту необхідно виконати завдання одної частини роботи на MASM32, а другої частини – на Visual Studio

Завдання для роботи на MASM32

Усім тудентам необхідно створити і налагодити у середовищі MASM32 програму **Lab1.** Це буде перша з двох програм лабораторної роботи №1.

Варіант завдання у даному випадку – це прізвище, ім'я та по-батькові студента (П.І.Б.).

Необхідно запрограмувати, щоб разом із попереднім текстом вітання у цьому ж діалоговому вікні програми виводилася б інформація (П.І.Б.) про студента як автора програми.

Примітка. Завдання можна виконати і в середовищі MASM64 – оцінка може бути збільшена, якщо студент пояснить та продемонструє особливості 64-бітових програм

Завдання для роботи на Visual Studio

Усім студентам також необхідно створити у середовищі Visual Studio програму **Lab1_cpuid.** Ця програма присвячена дослідженню команд MOV та CPUID і має виконувати

- показ початкового діалогового вікна-вітання від автора програми;
- команди CPUID з параметрами 0, 1, 2 а також 80000000h, 80000001h, 80000002h, 80000003h, 80000004h, 80000005h та 80000008h. Кожний результат виконання CPUID команди потрібно виводити у окремому діалоговому вікні. Якщо результати CPUID утворюють текстові дані, то виводити їх як рядки тексту.

Отримати дизасемблерний код і проаналізувати його.

Пояснити значення *N*-го біту кожного результату команди CPUID, де *N* – номер студента у журналі. Для пояснення використати документ "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference". Треба знайти, показати цей документ а також продемонструвати, як знайти у ньому потрібні відомості відповідно варіанту завдання.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми, основних результатів. Скріншоти виконання програм **Lab1** та **Lab1_cpuid**.
4. Аналіз, коментар вихідного тексту та дизасемблерного коду
5. Висновки

Контрольні запитання

1. Що означає **.386, .model flat, include** та **includelib**?
2. Що означає директива **DB**?
3. Де точка входу у програму?
4. Що таке **invoke**?
5. Які аргументи у функції **MessageBox**?
6. Навіщо потрібна функція **ExitProcess**?
7. Що роблять компілятор та лінкер? Як отримати виконуваний файл?
8. Як додати файл вихідного тексту на асемблері у проект MS Visual Studio?
9. Як налаштовувати конфігурації Debug та Release?
10. Як простежити роботу програми у налагоджувачі MS Visual Studio?
11. Що таке дизасемблерний код?
12. Як задати параметр функції команди **CPUID**?

Лабораторна робота №2. Створення модульних проектів на Асемблері та вивчення форматів представлення чисел

Мета: Навчитися створювати модульні проекти на Асемблері, а також закріпити знання основних форматів представлення чисел у комп’ютері.

Завдання

1. Створити у середовищі Microsoft Visual Studio проект з ім’ям **Lab2**.
2. Написати вихідний текст програми згідно варіанту завдання. Вихідний текст повинен бути у вигляді двох модулів на асемблері:

   - головний модуль, у якому описується загальний хід виконання програми від початку і до завершення. Цей модуль містить точку входу у програму, впродовж роботи викликає процедури з інших модулів. Вихідний текст головного модуля записати у файл **main2.asm**;
   - другий модуль, який містить процедуру, яка викликається з головного модуля. Цей модуль записати у файл **module.asm**.
3. Додати файли модулів у проект. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуваний файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати та вихідний текст.

Теоретичні відомості

Позиційна система числення

У типовій позиційній системі числення деяке число A представляється множиною цифр: $a_{n-1}\ a_{n-2}\ \ldots\ a_1\ \ a_0$. Числове значення A можна знайти по формулі:

$$A = a_{n-1}\, r^{n-1} + \ldots + a_2 r^2 + a_1 r + a_0, \qquad (2.1)$$

де r – основа системи числення,

n – розрядність,

a_i – цифри числа, зазвичай дорівнюють 0, 1, … , r-1. Позиція кожної цифри означає показник степені r.

У комп'ютері для представлення чисел використовується двійкова (r=2) система. Двійкові коди використовуються для виконання операцій та обміну інформації між пристроями комп'ютера. Проте програміст на асемблері може використовувати як двійкову, так і більш звичну для людині десяткову (r=10), а також вісімкову (r=8) та шістнадцяткову (r=16) системи.

Звичайний двійковий код, який описується формулою (2.1), використовується для представлення n-бітових позитивних чисел (без знаку).

Доповняльний код

Доповняльний n-розрядний код числа A можна описати у такий спосіб:

$$[A]_{дк} = A \quad (якщо\ A \geq 0) \qquad (2.2)$$
$$= r^n - |A| \quad (якщо\ A < 0)$$

якщо вважати, що для представлення |A| достатньо (n-1) розрядів звичайного беззнакового коду. Результат віднімання r^n - $|A|$ дає число з цифрами, які позначимо d_i. Для від'ємних чисел цифра лівого (старшого) розряду двійкового доповняльного коду дорівнює 1:

```
              1    0    0    0    . . . 0    0       (2^n)
        -
                        a_n-2 a_n-3 . . . a_1  a_0    (|A|)
               ------------------------------
[A<0]_дк  =         1    d_n-2 d_n-3 . . . d_1  d_0
```

Двійковий доповняльний код позитивних чисел утворюється так: до наявних (n-1) бітів дописується зліва 0:

```
[A>=0]_дк =         0    a_n-2 a_n-3 . . . a_1  a_0
```

Старший розряд називають **знаковим** – його цифра прямо вказує на знак числа (якщо не було переповнення розрядної сітки після виконання деякої операції). При виконання операцій додавання та віднімання чисел у доповняльному коді знакові розряди обробляються так само, як інші розряди.

Для перетворення у доповняльний код від'ємного числа (A< 0) у двійковій системі замість віднімання можна скористатися таким алгоритмом:

1. Спочатку дописується 0 зліва до розрядів числа |A|;
2. Виконується порозрядна інверсія всіх бітів;
3. Додається +1 в молодший розряд.

Формати представлення цілих чисел

Для роботи з цілими числами у архітектурі x86 використовуються формати розрядністю 8-біт (байт), 16-біт (слово), 32-біт (подвійне слово), 64-біт (квадрослово). У кожному з цих форматів передбачена робота як з числами без знаку, так і з числами зі знаком. Числа без знаку кодуються звичайним двійковим кодом. Для чисел зі знаком використовується доповняльний код.

Формати з плаваючою точкою

Математичний опис двійкового числа у форматі з плаваючою точкою:

$$V = (-1)^S 2^E \cdot M, \tag{2.3}$$

де S – знак, E – експонента, M – мантиса. Мантиса (M) складається з цілої та дробової частини. Дробова частина позначається як F.

Процесори сімейства x86, як і багато інших процесорів, підтримують двійкові формати з плаваючою точкою відповідно стандарту IEEE 754.

Одинарний 32-бітовий двійковий формат з плаваючою точкою

У 32-бітовому форматі, який у стандарті IEEE 754 зветься як *Single Precision,* є один біт (S) для знаку, 8 бітів (e) для кодованої експоненти та 23 бітів (F) для дробової частини мантиси (рис. 2.1)

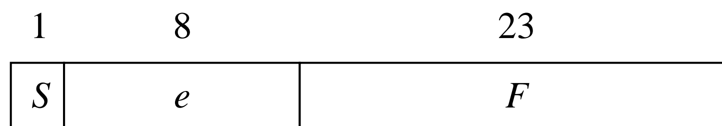


Рис. 2.1. Компоненти 32-бітового двійкового формату з плаваючою точкою

Експонента записується у зміщеному коді: $e = E+127$. Діапазон для експоненти E: від $E_{min} = -126$ до $E_{max}=+127$. Виходячи з цього, можна оцінити діапазон представлення чисел: ±(від 2^{-126} до 2^{+127}).

Мантиса має вигляд $M = 1.F$, тобто ціла частина завжди дорівнює одиниці. Оскільки це заздалегідь відомо, то для заощадження пам’яті біт цілої частини в пам’ять не записується – цей біт зветься "схованим бітом".

Розглянемо приклад представлення числа у цьому форматі. Спробуємо записати число π (а точніше кажучи, його приблизне значення), надане 30 десятковими розрядами

π = **3.14159265358979323846264338327...**

У двійковому 32-бітовому форматі з плаваючою точкою отримаємо наступне:

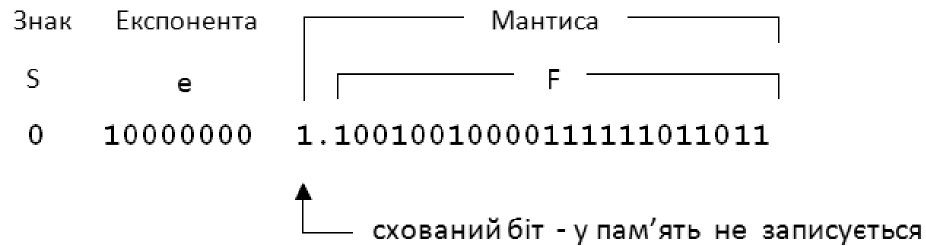


Рис. 2.2. Представлення значення π у 32-бітовому форматі

Для зберігання числа у пам'яті, починаючи з деякої адреси, буде записано чотири байти (враховуючи порядок байтів, прийнятий у процесорах Intel):

```
Адреса + 3     01000000   (знак та старші біти експоненти)
Адреса + 2     01001001
Адреса + 1     00001111
Адреса числа   11011011   (молодші біти мантиси)
```

Яка точність представлення чисел у цьому форматі? Отриманий вище двійковий код насправді означає зовсім інше число, аніж було потрібно.

```
Потрібно: 3.14159265358979323846264338327

Отримали: 3.14159274101257318400000000000
```

Відмінності починаються з сьомого десяткового розряду. Точність представлення обумовлюється розрядністю мантиси у 24 біти. Відносну похибку можна оцінити як 2^{-24}. Іншими словами, це означає точність 6-7 десяткових розрядів.

Подвійний 64-бітовий формат з плаваючою точкою

Цей формат, який у стандарті IEEE 754 зветься *Double Precision,* складається з наступних компонент

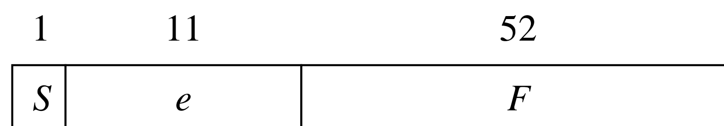


Рис. 2.3. Компоненти 64-бітового двійкового формату з плаваючою точкою

Кодоване значення експоненти: $e = E + 1023$. Експонента E може приймати значення від $E_{min} = -1022$ до $E_{max}=+1023$, тому загальний діапазон представлення чисел: ±(від 2^{-1022} до 2^{+1023}).

Мантиса записується так само, як і у 32-бітовому форматі: $M = 1.F$. Враховуючи схований біт цілої частини, мантиса загалом має 53 біти.

Приклад запису числа у подвійному форматі. У цьому форматі розглянуте вище число

3.14159265358979323846264338327

кодується вже наступним чином:

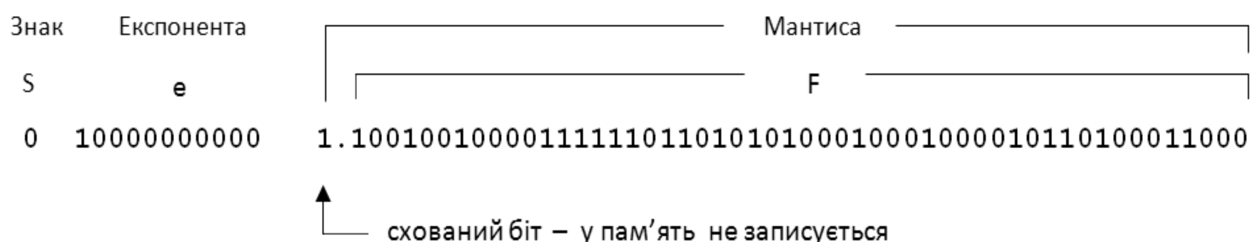


Рис. 2.4. Представлення значення π у 64-бітовому форматі

У пам'яті записується вісім байтів:

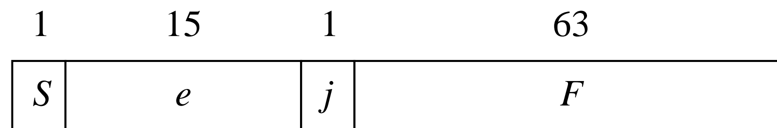
```
Адреса + 7     01000000
Адреса + 6     00001001
Адреса + 5     00100001
Адреса + 4     11111011
Адреса + 3     01010100
Адреса + 2     01000100
Адреса + 1     00101101
Адреса числа   00011000
```

Оцінимо точність подвійного формату.

```
Потрібно: 3.14159265358979323846264338327

Отримали: 3.14159265358979328000000000000
```

Тут відмінності починаються з 18-го розряду. Оскільки подвійний формат має 53-бітову мантису, то оцінимо відносну похибку як 2^{-53}. Це відповідає 17-18 десятковим розрядам.

Розширений 80-бітовий формат з плаваючою точкою



1	15	1	63
S	*e*	*j*	*F*

Рис. 2.5. Компоненти 80-бітового двійкового формату з плаваючою точкою

Кодоване значення експоненти: е = E + 16383. Діапазон експоненти E: від E_{min} = -16382 до E_{max}=+16383. Діапазон представлення для нормалізованих чисел: ±(від 2^{-16382} до 2^{+16383}).

У цьому форматі на відміну 32-, та 64-бітових форматів, мантиса записується повністю – у вигляді

$$M = j.F,$$

де j – це біт цілої частини, який може бути 0 або 1. Цей біт записується у пам'ять, як і решта бітів. Це зроблено тому, що у розширеному форматі передбачено виконання операцій також і над ненормалізованими числами, у яких старші біти мантиси можуть бути нулями.

Методичні рекомендації

Процес створення програми, як правило – це послідовність багатьох кроків невеличких змін. У процесі доведення розробки до бажаного результату часто корисно рухатися шляхом поступового розвитку, щоб у кожний поточний момент мати хоч і не завершену, проте працюючу версію програми. Не варто намагатися одразу все написати, а потім налагоджувати та шукати помилки в громіздкому тексті розкиданому по багатьом модулям. Однією з рекомендацій по створенню нової модульної програми буде така: включайте у проект не одразу усі модулі, а по одному, компілюючи проект після кожного додавання чергового модуля.

1. Створіть у середовищі MS Visual Studio новий проект з ім'ям **Lab2**. Як створювати подібний проект програми на асемблері – про це докладно написано у поясненнях до попередньої лабораторної роботи №1.

2. Додайте у проект порожній файл з ім'ям **main2.asm**. Цей файл буде головним файлом програмного коду. Для спрощення виконання роботи скористайтеся текстом головного файлу *.asm попередньої роботи №1. Скопіюйте текст і у вікні редагування вихідного тексту вилучіть зайві рядки.

Запишіть на диск головний файл програми **main2.asm**, у якому поки що тільки скелет, шаблон асемблерного коду програми, яка поки що нічого не робить.

3. Розпочніть компіляцію, якщо виникли помилки – виправте їх. Після успішної компіляції викличте програму на виконання. Закрийте її. Тепер можна доповнювати скелет змістовним кодом.

4. Додайте у проект модуль з ім'ям **module.asm**. Файл module.asm містить текст процедури StrHex_MY і наданий нижче:

```
.586
.model flat, c
.code
;процедура StrHex_MY записує текст шістнадцятькового коду
;перший параметр - адреса буфера результату (рядка символів)
;другий параметр - адреса числа
;третій параметр - розрядність числа у бітах (має бути кратна 8)
StrHex_MY proc
   push ebp
   mov ebp,esp

   mov ecx, [ebp+8]          ;кількість бітів числа
   cmp ecx, 0
   jle @exitp
   shr ecx, 3                ;кількість байтів числа
   mov esi, [ebp+12]         ;адреса числа
   mov ebx, [ebp+16]         ;адреса буфера результату
@cycle:
   mov dl, byte ptr[esi+ecx-1]   ;байт числа - це дві hex-цифри

   mov al, dl
   shr al, 4                 ;старша цифра
   call HexSymbol_MY
   mov byte ptr[ebx], al

   mov al, dl                ;молодша цифра
   call HexSymbol_MY
   mov byte ptr[ebx+1], al

   mov eax, ecx
   cmp eax, 4
   jle @next
   dec eax
   and eax, 3                ;проміжок розділює групи по вісім цифр
   cmp al, 0
   jne @next
   mov byte ptr[ebx+2], 32   ;код символа проміжку
   inc ebx

@next:
```

```
   add ebx, 2
   dec ecx
   jnz @cycle
   mov byte ptr[ebx], 0       ;рядок закінчується нулем
@exitp:
   pop ebp
   ret 12
StrHex_MY endp

;ця процедура обчислює код hex-цифри
;параметр - значення AL
;результат -> AL
HexSymbol_MY proc
   and al, 0Fh
   add al, 48         ;так можна тільки для цифр 0-9
   cmp al, 58
   jl @exitp
   add al, 7          ;для цифр A,B,C,D,E,F
@exitp:
   ret
HexSymbol_MY endp

end
```

Окрім файлу **module.asm** потрібно у робочу папку проекту записати файл заголовку модуля **module.inc**. У файлі заголовку вказуються директивою EXTERN імена процедур, які можуть викликатися іншими модулями. Поки що тільки одна така процедура у цьому модулі, тому файл **module.inc** містить один рядок:

```
EXTERN StrHex_MY : proc
```

5. Після того, як у проект додали файл вихідного тексту **module.asm**, скомпілюйте проект і викличте програму на виконання. Якщо усе гаразд, то можна продовжити наповнювати головний файл **main3.asm** потрібним змістом.

6. Як викликати процедуру StrHex_MY? Для цього спочатку потрібно у тексті **main3.asm** записати рядок

```
include module.inc
```

Тепер можна використовувати процедуру StrHex_MY. Щоб її правильно викликати, необхідно передати потрібні параметри. Які параметри – про це слід прочитати у коментарі в тексті файлу **module.asm**. Надамо приклад виклику процедури StrHex_MY:

```
push offset TextBuf
push offset Value
push 32
call StrHex_MY
```

У цьому випадку процедура оброблятиме значення 32-бітової перемінної Value і повинна записати результат у масив TextBuf.

Потім потрібно показати результат – вміст масиву TextBuf. Для цього можна скористатися вже відомим діалоговим вікном MessageBox:

```
invoke MessageBoxA, 0, ADDR TextBuf, ADDR Caption, MB_ICONINFORMATION
```

Потрібно створити масиви для двох рядків тексту – Caption та TextBuf. Масив Caption вже був у текстах попередньої роботи №1. Масив для рядку тексту можна створити у програмі, наприклад, так:

```
TextBuf db 64 dup(?)
```

Можна вважати, що текстовий буфер для максимум 63 символів буде цілком придатним для виконання цієї лабораторної роботи. Перевірте це.

Примітка. При виклику діалогового вікна MessageBox вище був вказаний параметр `MB_ICONINFORMATION` – іменована константа з API Win32. Рекомендується підключити заголовочний файл **windows.inc** у такий спосіб:

```
option casemap :none        ;розрізнювати великі та маленькі букви

include \masm32\include\windows.inc

. . .                       ;інші include та includelib
```

7. Щоб запрограмувати дослідження деякого числового значення у 32-бітовому форматі з плаваючою точкою, для цього можна створити 32-бітну перемінну з ім’ям Value, ініціалізувавши її потрібним значенням, наприклад:

```
Value dd 3.14159265358979323846264338327
```

Після виклику процедур StrHex_MY і MessageBoxA отримаємо такий результат:

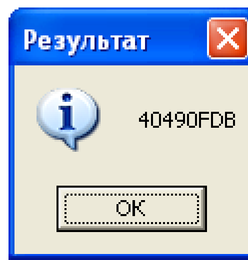


Рис. 2.6. Шістнадцятковий код 32-бітового числа

Кожна шістнадцяткова цифра відповідає 4 бітам двійкового коду. З восьми символів відтворюється 32-бітний код числа з плаваючою точкою:

```
 4    0    4    9    0    F    D    B

0100 0000 0100 1001 0000 1111 1101 1011
```

Порівняйте отриманий код із двійковим кодом для π, наведеним у теоретичних відомостях для цієї лабораторної роботи.

Після отримання двійкового коду числа у форматі з плаваючою точкою потрібно виділити біт знаку, біти кодованої експоненти та біти мантиси.

Окрім чисел у форматах з плаваючою точкою, програма повинна виводити шістнадцяткові коди і для цілих чисел згідно завданню.

Для того, щоб отримати значення інших типів, можна створювати перемінні за допомогою відповідних директив: DB для оголошення 8-бітових перемінних, DW – для 16-бітових, DD – для 32-бітових, DQ – для 64-бітових, DT – для 80-бітових. При створенні кожної перемінної ініціалізувати її потрібним значенням.

Варіанти завдань та основні вимоги

Номер варіанту (N) згідно списку студентів у журналі. Студент виконує завдання для числових значень X та Y, які обчислюються за формулами:

$$X = N+10$$
$$Y = 2X$$

Студент має запрограмувати на асемблері вивід шістнадцяткових значень для усіх типів даних згідно наведеній нижче таблиці. У звіті надати таблицю, заповнену кодами-результатами.

Таблиця 2.1

Значення кодованих чисел

Типи даних, які має обробити програма і показати кодовані значення	Значення	Як потрібно надати у звіті результати виконання програми	
Ціле 8-бітове	X	шістнадцятковий код	двійковий код
	-X	шістнадцятковий код	двійковий код
Ціле 16-бітове	X	шістнадцятковий код	двійковий код
	-X	шістнадцятковий код	двійковий код
Ціле 32-бітове	X	шістнадцятковий код	двійковий код
	-X	шістнадцятковий код	двійковий код
Ціле 64-бітове	X	шістнадцятковий код	двійковий код
	-X	шістнадцятковий код	двійковий код
Число у 32-бітовому форматі з плаваючою точкою	X.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	-Y.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	X.X	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
Число у 64-бітовому форматі з плаваючою точкою	X.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	-Y.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	X.X	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
Число у 80-бітовому форматі з плаваючою точкою	X.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	-Y.0	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)
	X.X	шістнадцятковий код	двійковий код (вказати біт знаку, біти експоненти, біти мантиси)

Примітка. Двійковий код можна отримати і без комп'ютера, перетворивши "вручну" або на калькуляторі відповідні шістнадцяткові коди результатів. Але, якщо студент при виконанні лабораторної роботи

запрограмує окремою процедурою також і вивід двійкових кодів – оцінка буде підвищена.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів та вихідного тексту
6. Висновки

Контрольні запитання

1. Як створюється 8-бітова перемінна?
2. Як створюється 16-бітова перемінна?
3. Як створюється 32-бітова перемінна?
4. Як створюється 64-бітова перемінна?
5. Як створюється 80-бітова перемінна?
6. Що таке доповняльний код?
7. Що таке експонента числа з плаваючою точкою?
8. Що таке мантиса?
9. Від чого залежіть діапазон представлення чисел у форматах з плаваючою точкою?
10. Від чого залежіть точність представлення чисел у форматах з плаваючою точкою?
11. Що означає **push offset**?
12. Що означає **call**?

Лабораторна робота №3. Програмування арифметичних операцій підвищеної розрядності

Мета: Навчитися програмувати на асемблері основні арифметичні операції підвищеної розрядності, а також отримати перші навички програмування власних процедур у модульному проекті.

Завдання

1. Створити у середовищі Microsoft Visual Studio проект з ім'ям **Lab3**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main3.asm**. Цей модуль створити та написати заново, частково використавши текст модуля main2.asm попередньої роботи №2;
   - другий модуль: використати **module** попередньої роботи №2;
   - третій модуль: створити новий з ім'ям **longop**.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуємий файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати, вихідний текст та дизасемблерний машинний код програми.

Теоретичні відомості

Поняття "підвищена розрядність" у даній роботі означає таку кількість біт, яку процесор не може обробити одною командою. Наприклад, у процесорах 32-бітової архітектури арифметичні команди виконуються для 32-бітових чисел, і таку саму розрядність мають робочі регістри (EAX, EBX, ECX, EDX тощо). Процесор 64-бітовий здатен одною командою додавати вже 64-бітові числа. Проте, може виникнути потреба у роботі з числовими даними, розрядність яких може суттєво – у десятки разів перевищувати довжину машинного слова. Підвищена розрядність означає підвищену точність та (або) більший діапазон числових даних.

Реалізувати арифметику підвищеної розрядності можна власноруч програмним шляхом. Мова асемблера дозволяє це реалізувати достатньо просто та ефективно.

Додавання

Нехай потрібно додати 128-бітові числа. Представимо кожне 128-бітове число чотирма 32-бітовими групами: A_i та B_i . Спочатку треба додати молодші групи бітів: A_0+B_0 . Результат додавання буде 33-бітовим – 32 молодші біти та перенос. Молодші 32 біти вже є бітами результату, а перенос треба додати при обчисленні суми наступних груп: A_1+B_1+перенос. І так далі. Біт переносу результату додавання старших груп A_3+B_3 потрібно зберегти, якщо у цьому є потреба. Алгоритм додавання наведений на рис. 3.1.

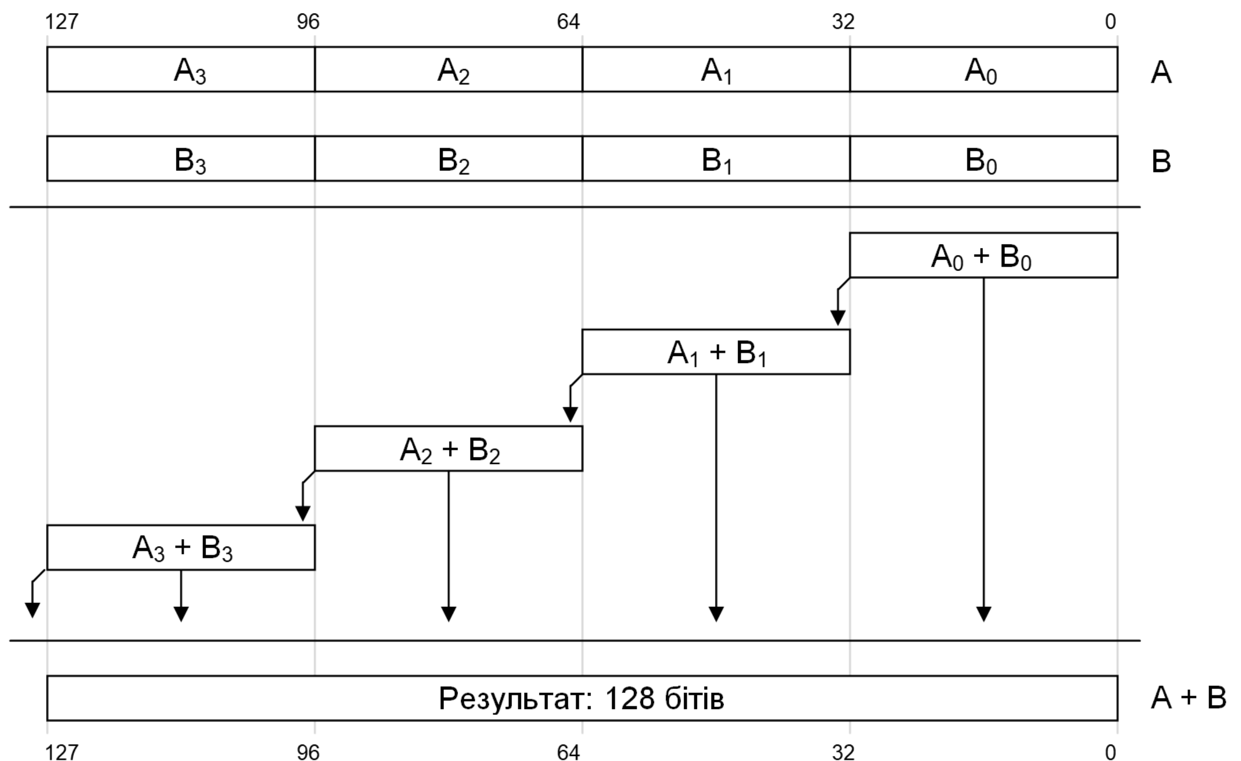


Рис. 3.1. Алгоритм додавання 128-бітових чисел групами по 32 біти

Такий алгоритм коректно працює як з числами без знаку, так і з числами зі знаком у доповняльному коді.

При реалізації додавання у програмах на асемблері використовується команда ADD для додавання молодших груп бітів A_0+B_0, і команда ADC – для додавання у вигляді: A_i+B_i+перенос. При виконанні додавання значення переносу автоматично записується у біт CF регістру EFLAGS.

Віднімання

Віднімання виконується так само, як і додавання – послідовно від молодших груп бітів. При відніманні наступних груп бітів потрібно враховувати результат віднімання попередньої групи.

Розглянемо віднімання на прикладі. Нехай потрібно від нуля відняти одиницю:

```
00..00 00..00 00..00 00..00   A
00..00 00..00 00..00 00..01   B
---------------------------
                     11..11   A0-B0
              11..11          A1-B1-позичання
       11..11                 A2-B2-позичання
11..11                        A3-B3-позичання
---------------------------
11..11 11..11 11..11 11..11   Результат A-B
```

Рис. 3.2. Приклад віднімання чисел групами бітів

Результат віднімання записується у доповняльному коді. У розглянутому вище прикладі це доповняльний код числа (-1).

Позичання виникає при відніманні більшого числа від меншого. Позичання означає вимогу розповсюдити групу одиниць від старшого до поточного біту. Цю групу одиниць достатньо закодувати одним бітом, який зветься бітом позичання (borrow).

При реалізації у програмах на асемблері для віднімання молодших груп бітів ($A_0 - B_0$) використовується команда SUB, а віднімання усіх наступних груп бітів ($A_i - B_i$) робиться командою SBB (subtract with borrow). Біт позичання записується у біт CF регістру EFLAGS.

Щодо програмної реалізації на асемблері додавання та віднімання

Програмний код можна оформити у вигляді процедур, які викликаються для виконання відповідної операції. Рекомендується надати процедурам імена, які позначають їхнє функціональне призначення, а також приналежність певному модулю, наприклад:

`Add_128_LONGOP` – процедура додавання 128-бітових даних;
`Sub_128_LONGOP` – процедура віднімання 128-бітових даних;

Кожна з таких процедур буде мати три параметри: операнд A, операнд B, та результат. При розробці будь-якої процедури, потрібно спочатку узгодити, що і як передавати процедурі у якості параметрів і у якому вигляді і як отримувати результат. У якості, наприклад, 128-бітових операндів та результату можна визначити масиви з чотирьох 32-бітових подвійних слів, прямо ініціалізувавши їх деякими числовими значеннями:

```
ValueA dd 80010001h, 80020001h, 80030001h, 80040001h
ValueB dd 80000001h, 80000001h, 80000001h, 80000001h
Result dd 4 dup(0)
```

Для процедур такі параметри зручно передавати у вигляді вказівників, адрес цих масивів. Наприклад:

```
push offset ValueA
push offset ValueB
push offset Result
call Add_128_LONGOP
```

Таким чином, три параметри процедурі передаються через стек. Процедура повинна прочитати значення з масивів ValueA, ValueB і записати результат у масив Result по вказаній адресі. Процедура сама повинна відновити стек. Такий програмний код буде для модуля-користувача процедури **Add_128_LONGOP.** Так само можна зробити для процедури віднімання.

Після такого зовнішнього визначення інтерфейсу обміну інформацією із процедурами, тепер можна переходити до програмування внутрішньої реалізації самих процедур. Фрагмент можливого варіанту реалізації процедури додавання двох 128-бітових чисел A та B:

```
Add_128_LONGOP proc
   push ebp
   mov ebp,esp

   mov esi, [ebp+16]            ;ESI = адреса A
   mov ebx, [ebp+12]            ;EBX = адреса B
   mov edi, [ebp+8]             ;EDI = адреса результату

   mov eax, dword ptr[esi]      ;починаємо з молодших груп
   add eax, dword ptr[ebx]      ;додавання 32-біт
   mov dword ptr[edi], eax      ;запис молодших 32 біт суми

   mov eax, dword ptr[esi+4]    ;наступна 32-бітова група
```

```
   adc eax, dword ptr[ebx+4]      ;додавання з урахуванням
                                  ;переносу з попередньої групи

   mov dword ptr[edi+4], eax      ;запис наступних 32 біт суми
   mov eax, dword ptr[esi+8]      ;наступна 32-бітова група
   adc eax, dword ptr[ebx+8]      ;додавання з урахуванням
                                  ;переносу з попередньої групи

   mov dword ptr[edi+8], eax      ;запис наступних 32 біт суми
   . . .                          ;і так далі

   pop ebp                        ;відновлення стеку
   ret 12
Add_128_LONGOP endp
```

Додавання багаторозрядних чисел тут запрограмовано послідовним додаванням 32-бітових груп, починаючи з групи молодших розрядів. Для додавання наймолодших розрядів виконується команда ADD. В результаті може виникнути перенос у наступну старшу групу. Цей перенос записується у біт CF регістру EFLAGS. Для додавання наступної 32-бітової групи бітів з урахуванням можливого переносу використана команда ADC. І так далі, до найстаршої 32-бітової групи.

Команда RET означає завершення роботи процедури. Тут ця команда також звільнює стек від 3 чотирибайтових параметрів.

У подібний спосіб можна запрограмувати процедуру **`Sub_128_LONGOP`**, призначену для віднімання 128-бітних даних. Обробку виконувати також 32-бітними групами, починаючи з молодших. Віднімання наймолодших 32 біт робити командою SUB, а наступні 32-бітові групи обробляти командою SBB.

Можна звернути увагу на те, що у вихідному тексті процедури додавання декілька разів повторюється майже однаковий код з команд MOV та ADC:

```
mov eax, dword ptr[esi+Зсув]      ;наступна 32-бітова група
adc eax, dword ptr[ebx+Зсув]      ;додавання з урахуванням переносу
                                  ;з попередньої групи
mov dword ptr[edi+Зсув], eax      ;запис наступних 32 біт суми
```

Значення зсуву – 0, 4, 8, і так далі, відповідно розрядності операції. Для зменшення обсягу вихідного тексту можна запрограмувати циклічне повторення цих команд з відповідним змінним значенням зсуву. При програмуванні циклу, потрібно враховувати необхідність зберігання біту

переносу CF – або якось зберігати цей біт, або для організації циклу використовувати команди, які б не змінювали біт переносу.

Програмування циклів

Цикл є дуже популярним елементом різноманітних алгоритмів. Як запрограмувати цикл на асемблері?

Нехай потрібно щось виконати 10 разів. Це "щось" будемо називати "тілом циклу". Тіло циклу може містити деяку множину рядків програмного коду на асемблері. Розглянемо варіант алгоритму циклу з постумовою

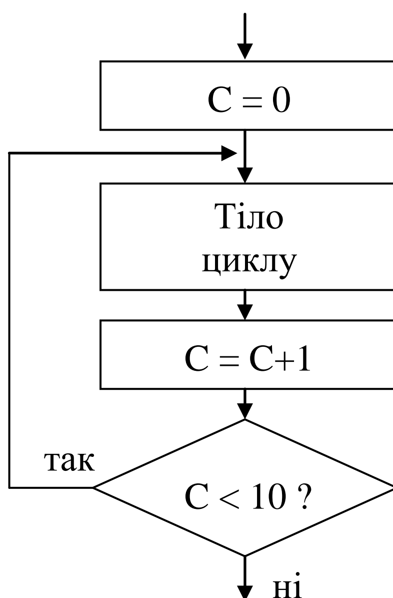


Рис. 3.2. Блок-схема алгоритму циклу з постумовою

На асемблері такий цикл можна запрограмувати наступним чином:

```
  mov ecx, 0        ; обнулюємо лічильник – регістр ECX
cycle:
  . . .             ; тіло циклу
  inc ecx           ; збільшуємо лічильник на 1
  cmp ecx, 10       ; порівнюємо лічильник з 10
  jl cycle          ; якщо лічильник менше – перехід на мітку cycle
```

Необхідно відзначити, що такий варіант організації циклу є неприйнятний для послідовної обробки груп бітів з розповсюдженням переносу, оскільки команда CMP сама записує біт CF – тобто знищує потрібне значення переносу, яке треба було б розповсюджувати.

Інший варіант циклу – у лічильник спочатку записується потрібна кількість повторень, а на кожній ітерації значення лічильника зменшується на одиницю:

```
  mov ecx, 10         ; у регістр ECX записуємо кількість повторень
cycle:
  . . .               ; тіло циклу
  dec ecx             ; зменшуємо лічильник на 1
  jnz cycle           ; якщо лічильник не 0, то перехід на мітку cycle
```

Такий варіант циклу може бути використаний для наших цілей, оскільки ані команда DEC, ані JNZ не змінюють біт CF регістру EFLAGS.

Як вже вказувалося вище, при додаванні чисел підвищеної розрядності виконується обробка 32-бітових за такою схемою: спочатку командою ADD додаються молодші 32 біти, а решта груп бітів додаються командою ADC. Для того, щоб запрограмувати це у циклі, команди тіла циклу повинні бути однаковими. Тіло циклу можливо побудувати на основі команд ADC, якщо попередньо обнулити біт переносу командною CLC:

```
  mov ecx, ...       ; ECX = потрібна кількість повторень
  clc                ; обнулює біт CF регістру EFLAGS
cycle:
  mov eax, dword ptr[esi+Зсув]
  adc eax, dword ptr[ebx+Зсув]        ; додавання групи з 32 бітів
  mov dword ptr[edi+Зсув], eax
  . . .              ; модифікація зсуву
  dec ecx            ; лічильник зменшуємо на 1
  jnz cycle
```

Для кожної ітерації потрібно сформувати адресу поточної групи бітів, наприклад, адреса

```
dword ptr[esi+Зсув]
```

складається з вмісту регістру ESI та зсуву. Зсув можна зберігати у якомусь регістрі, наприклад, EDX і при переході до наступної ітерації збільшувати його на одиницю:

```
  mov edx, 0
  . . .
cycle:
```

```
mov eax, dword ptr[esi+4*edx]
. . .
inc edx
dec ecx
jnz cycle
```

Необхідно відзначити, що наведені тут відомості не вичерпують усіх можливостей вирішення подібних завдань.

Методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім’ям **Lab3**.

2. Додайте у проект порожній файл з ім’ям **main3.asm**. Цей файл буде головним файлом програмного коду. Для спрощення виконання роботи скористайтеся текстом головного файлу *.asm попередньої роботи №2. Скопіюйте текст і у вікні редагування вихідного тексту вилучіть зайві рядки. Запишіть на диск головний файл програми **main3.asm**.

3. Додайте у проект модуль з ім'ям **module**. У проекті використовується файли **module.asm, module.inc** попередньої роботи №2 без будь-яких змін. Рекомендація: для того, щоб у декількох проектах використовувати ті самі модулі, запишіть файли цих модулів у окрему папку. Кожний файл вихідного тексту модулів, які спільно використовується, повинен бути у одному екземплярі.

4. Додайте у проект новий модуль з ім'ям **longop**. Файл вихідного тексту **longop.asm** буде містити вихідний текст процедур, які реалізують арифметичні операції підвищеної розрядності. Окрім файлу **longop.asm** потрібно створити файл заголовку **longop.inc**. Цей файл міститиме імена процедур, які будуть викликатися з інших модулів. Приклад вмісту файлу **longop.inc**:

```
EXTERN Add_128_LONGOP : proc
EXTERN Sub_128_LONGOP : proc
```

5.Програмування числових значень операндів із розрядністю згідно завданню. У файлі **main3.asm** потрібно запрограмувати створення перемінних потрібної розрядності, ініціалізувавши їх числовими значеннями згідно варіанту завдання.

6. Запрограмувати вивід результатів виконання операцій для кожного набору значень у окремому діалоговому вікні MessageBox.

7. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio. Відомості та методичні рекомендації надані у відповідних розділах попередніх лабораторних робіт №1, 2.

Варіанти завдань та основні вимоги

Номер варіанту (N) згідно списку студентів у журналі.

Кожному студенту необхідно запрограмувати на асемблері два додавання (A+B) та два віднімання (A-B).

Операнди A та B для першого додавання наступні:

```
A = 80010001h, 80020001h, 80030001h, 80040001h, ...
B = 80000003h, 80000003h, 80000003h, 80000003h, ...
```

(у шістнадцятковому коді)

Операнди A та B для другого додавання:

```
A = (N 32-біт), (N+1 32-біт), (N+2 32-біт), (N+3 32-біт), ...
B = 80000003h, 80000003h, 80000003h, 80000003h, ...
```

де N – номер варіанту

Операнди для першого віднімання:

```
A = 0
B = (N 32-біт), (N+1 32-біт), (N+2 32-біт), (N+3 32-біт), ...
```

де N – номер варіанту

Операнди для другого відніманняі:

```
A = (N 32-біт), (N+1 32-біт), (N+2 32-біт), (N+3 32-біт), ...
B = 00000001h, 00000001h, 00000001h, 00000001h, ...
```

де N – номер варіанту

Примітка. Числові значення операндів тут вказані від молодших ліворуч до старших праворуч розрядів групами по 32 бітів. Записані явно перші 128 бітів, далі відповідно варіанту розрядності.

Вимоги щодо формуваня числових значень операндів перед виконанням операцій:

- для номерів варіантів N = 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, . . . операнди для **другого додавання** формувати у циклі, решту – без циклу

- для номерів варіантів N = 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, . . . операнди для **першого додавання** формувати у циклі, решту – без циклу
- для номерів варіантів N = 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, . . . операнди для **віднімання** формувати у циклі, решту – без циклу

Розрядність операндів A та B для кожного варіанту завдання – у таблиці

Таблиця 3.1

№ варіанту	Розрядність додавання (біт)	Розрядність віднімання (біт)
1	96	1024
2	128	992
3	160	960
4	192	928
5	224	896
6	256	864
7	288	832
8	320	800
9	352	768
10	384	736
11	416	704
12	448	672
13	480	640
14	512	608
15	544	576
16	576	544
17	608	512
18	640	480
19	672	448
20	704	416
21	736	384
22	768	352
23	800	320
24	832	288
25	864	256
26	896	224
27	928	192
28	960	160
29	992	128
30	1024	96

Примітка. Розрядності операндів та інші параметри варіантів завдань можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасемблерного машинного коду
6. Висновки

Контрольні запитання

1. Як виконується додавання підвищеної розрядності?
2. Як виконується віднімання підвищеної розрядності?
3. Як передаються параметри процедурам?
4. Якими командами виконується додавання та віднімання?
5. Що таке перенос і як він використовується?
6. Що таке позичання і як воно використовується?
7. Як запрограмувати цикл на асемблері?
8. Що виконують команди DEC, INC, JNZ, CLC?

Лабораторна робота №4. Програмування множення чисел підвищеної розрядності

Мета: Навчитися програмувати на асемблері множення чисел підвищеної розрядності, а також закріпити навички програмування власних процедур у модульному проекті.

Завдання

1. Створити у середовищі MS Visual Studio проект з ім'ям **Lab4**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main4.asm**. Цей модуль створити та написати заново, частково використавши текст модуля main3.asm попередньої роботи №3;
   - другий модуль: використати **module** попередніх робіт №2, 3;
   - третій модуль: модуль **longop** попередньої роботи №3 доповнити новим кодом відповідно завданню.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуваний файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати вихідний текст та результати.

Теоретичні відомості

Обчислення факторіалу

Факторіалом $n!$ зветься добуток цілих чисел від 1 до n

$$n! = 1 \cdot 2 \cdot 3 \cdot . . . \cdot (n-1) \cdot n$$

Факторіал часто використовується у різноманітних галузях математики. Наприклад, з комбінаторики відомо, що кількість усіх можливих перестановок n елементів дорівнює $n!$

Числові значення факторіалу стрімко зростають при збільшенні n. Це ускладнює розрахунки там, де потрібна велика точність. Можна розглянути декілька прикладів значень факторіалу (табл. 4.1).

Таблиця 4.1

Деякі значення факторіалу

n	$n!$ (приблизні значення для n>10)	Кількість бітів (N), потрібних для точного представлення $n!$
1	1	1
2	2	2
3	6	3
4	24	5
5	120	7
6	720	10
7	5040	13
8	40320	16
9	362880	19
10	3628800	22
20	$2.4329 \cdot 10^{18}$	62
30	$2.65253 \cdot 10^{32}$	108
40	$8.15915 \cdot 10^{47}$	160
50	$3.04141 \cdot 10^{64}$	215
60	$8.32099 \cdot 10^{81}$	273
70	$1.1979 \cdot 10^{100}$	333
80	$7.1569 \cdot 10^{118}$	395
90	$1.4857 \cdot 10^{138}$	459
100	$9.3326 \cdot 10^{157}$	525

Для обчислення приблизного значення факторіалу можна скористатися формулою Стірлінга. У першому наближенні оцінка має вигляд:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Алгоритм для точного обчислення факторіалу є дуже простим:

```
factorial = 1;
for (i=2; i<=n; i++)
   factorial *= i;
```

проте, щоб реалізувати точні обчислення, потрібно виконувати множення чисел великої розрядності. Наприклад, при обчисленні 100! потрібно на останньому кроці перемножувати 519-бітне значення щоб отримати 525-бітовий результат.

Взагалі, для представлення будь-якого цілого позитивного числа X потрібно не менше $N = (1+\log_2 X)$ двійкових розрядів.

Множення підвищеної розрядності N×N

У якості прикладу розглянемо множення двох 96-бітових операндів A та B. Результат буде мати удвічі більшу (2N) розрядність – 192 біти. Один з можливих алгоритмів виконання множення групами по 32 біт полягає у множенні одного 96-бітного операнду (A) на групи 32 бітів іншого операнду (B_j). У свою чергу, множення 96-бітного A на групу бітів B_j виконується за три кроки – на кожному кроці отримується 64-бітовий добуток двох 32-бітових груп. Для отримання результату потрібно додати усі 64-бітові добутки відповідно їхньому розташуванню у 192-бітовій розрядній сітці (рис.4. 1).

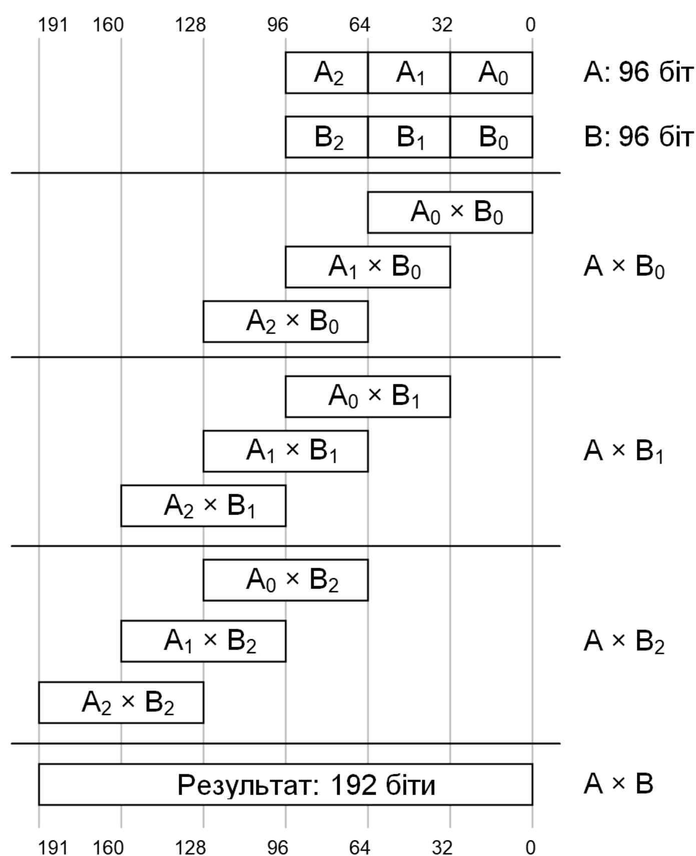


Рис. 4.1. Множення 96-бітових чисел групами по 32 біти

Опис алгоритму С-подібним псевдокодом:

```
Результат = 0;
for (j=0; j<n32; j++)
  for (i=0; i<n32; i++)
      Результат += A[i]*B[j];
```

де n32 – це кількість 32-бітових груп, тобто n32= N/32. Цей алгоритм коректно працює тільки для чисел без знаку.

Для обчислення часткових добутків ($A_i \times B_j$) у програмі на асемблері можна скористатися командою MUL – множення чисел без знаку.

```
mul src
```

Ця команда виконує множення операнда src на значення у регістрі AL, або AX, або EAX, або RAX у залежності від розрядності операнду src. Результат записується відповідно у регістр AX, або регістри AX:DX, або у регістри EAX:EDX, або у регістри RAX:RDX. Якщо операнд src є 32-бітовим, то результат множення EAX*src буде 64-бітовим: молодші 32 біти результату записуються у EAX, старші – у EDX.

Множення N×32

Розглянемо дещо скорочений варіант множення підвищеної розрядності – коли один з операндів повнорозрядний, а інший операнд 32-бітовий. Такий різновид множення може бути використаний, коли один з множників гарантовано представлятиметься не більш, як 32 бітами. Наприклад, при обчисленні факторіалу $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$, якщо $n < 2^{32}$. Порівняно із множенням N×N, множення N×32 є набагато простішим

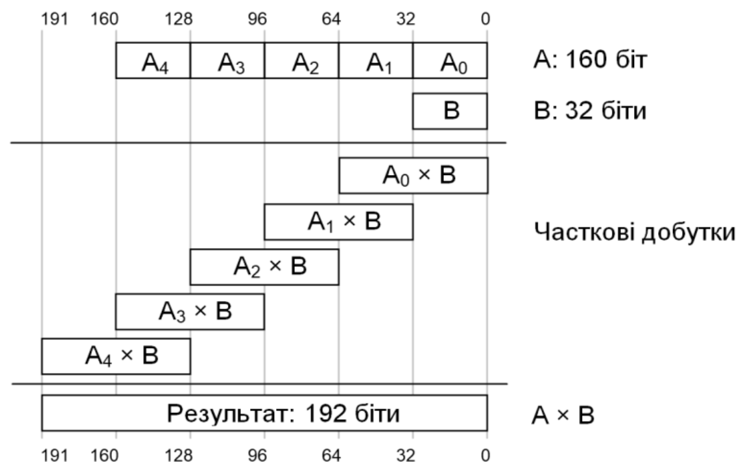


Рис. 4.2. Множення 160-бітового числа на 32-бітове

Можна також відмітити, що для N×32 є набагато простішим розповсюдження переносів при додаванні часткових добутків, аніж для множення N×N.

На основі алгоритму множення N×32 можливо побудувати процедуру, яка записує результат множення у операнд підвищеної розрядності. Це є зручним при організації ланцюжків обчислень, наприклад, при обчисленні факторіалу.

Деякі особливості програмування циклів

При програмуванні на асемблері таких операцій, як множення підвищеної розрядності, може виникнути проблема нестачі регістрів процесора. Для забезпечення високої швидкодії у якості лічильників циклів бажано використовувати регістри процесора, проте часто трапляється так, що регістрів загального призначення не вистачає. У цьому випадку для лічильників циклів залишається використовувати перемінні, розташовані у сегменті даних або у стеку. Це уповільнює роботу, особливо при виконанні великої кількості циклів, у тілі яких міститься мало команд.

Цикли можуть бути вкладеними, тому рекомендується використовувати регістри у першу чергу у внутрішньому циклі. Наприклад:

```
.data
  counter dd 0              ; ця перемінна буде лічильником циклу1
.code

@cycle1:                    ; початок зовнішнього циклу1
  mov eax, counter          ; завантажуємо значення перемінної
  inc eax                   ; збільшуємо лічильник на одиницю
  cmp eax, maxcycle1        ; порівнюємо з макс. значенням лічильника
  jg @exit                  ; вихід з циклу
  mov counter, eax          ; зберігаємо значення лічильника у пам'яті
  . . .                     ; тіло зовнішнього циклу1
  mov ecx, numcycle2            ; кількість повторень для циклу2
@cycle2:                        ; початок внутрішнього циклу2
  . . .                         ; тіло внутрішнього циклу2
  dec ecx                       ; зменшуємо лічильник циклу2 у регістрі ECX
  jnz @cycle2                   ; перехід на початок циклу2
  . . .                     ; тіло зовнішнього циклу1
  jmp @cycle1               ; перехід на початок циклу1
```

У наведеному вище прикладі є два цикли. У тілі першого циклу міститься вкладений цикл. Для вкладеного циклу лічильник у регістрі ECX. Лічильник для зовнішнього циклу зберігається у перемінній counter.

Методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім’ям **Lab4**.

2. Додайте у проект порожній файл з ім’ям **main4.asm**. Цей файл буде головним файлом програмного коду. Для спрощення виконання роботи скористайтеся текстом головного файлу *.asm попередньої роботи №3. Скопіюйте текст і у вікні редагування вихідного тексту вилучіть зайві рядки. Запишіть на диск головний файл програми **main4.asm**.

3. Додайте у проект модуль з ім'ям **module**. У проекті використовується файли **module.asm, module.inc** попередніх робіт №2, 3 без будь-яких змін. Рекомендація: для того, щоб у декількох проектах використовувати ті самі модулі, запишіть файли цих модулів у окрему папку. Кожний файл вихідного тексту модулів, які спільно використовується, повинен бути у одному екземплярі.

4. Додайте у проект модуль **longop**. У проекті використовуються файли **longop.asm**, **longop.inc** попередньої роботи №3. У ці файли повинні бути додані програмні коди процедур множення N×N та N×32. Для цього необхідно визначити розрядність (N), необхідну для представлення факторіалу (n!). Рекомендується N обирати кратною 32.

5. У файлі **main4.asm** потрібно запрограмувати цикл для обчислення значення факторіалу згідно варіанту завдання. Рекомендується у циклі обчислення факторіалу використати процедуру множення N×32. Отримане значення n! потім возвести у квадрат – перемножити за допомогою процедури множення N×N.

6. Запрограмувати множення тестових значень згідно варіанту завдання викликом процедур множення N×N та N×32.

7. Запрограмувати вивід результатів у діалоговому вікні MessageBox. Запрограмувати вивід потрібних числових значень у шістнадцятковому коді.

8. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio. Відомості та методичні рекомендації надані у відповідних розділах попередніх робіт.

Варіанти завдань та основні вимоги

Для кожного студента своє значення n, яке визначається за формулою $n = 30 + 2 \times H$, де H – це номер студента у журналі.

Потрібно запрограмувати на асемблері:

- обчислення факторіалу $\boldsymbol{n!}$
- обчислення квадрату факторіалу $\boldsymbol{n! \times n!}$
- обчислення тесту множення двійкових кодів 111…1×111…1 розрядністю операндів N×N. Розрядність кожного операнду (N) має бути тою самою, яка була обрана для представлення значення $\boldsymbol{n!}$ вашого варіанту. Шістнадцяткові коди таких операндів мають вигляд FF…F × FF…F.
- обчислення тесту множення двійкових кодів 111…1 × 111…1 розрядністю операндів N×32.
- обчислення тесту множення двійкових кодів 0101…0101 × 10…01 розрядністю операндів N×N. Шістнадцяткові коди таких операндів мають вигляд 55…5 × 80…01.

Точні значення результатів надати у шістнадцятковій системі числення.

Примітка. Вказані тести можуть бути викладачем замінені на інші тести спеціальним оголошенням перед видачею завдань студентам.

Оцінка може бути підвищена у таких випадках:

- розроблена процедура множення N×32, яка записує результат у операнд підвищеної розрядності, і ця процедура використана при обчисленні факторіалу $\boldsymbol{n!}$
- розроблена 64-бітова програма, у якій множення 64-бітовими порціями, зокрема, замість процедури розрядності N×32 – процедура N×64 (тоді N може бути кратною 64).

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми та результатів виконання програми
4. Аналіз, коментар вихідного тексту та результатів
5. Висновки

Контрольні запитання

1. Які проблеми виникають при програмуванні обчислення факторіалу?
2. Як оцінити значення факторіалу?
3. Як визначити потрібну розрядність для виконання операцій та представлення результатів?
4. Як виконується множення підвищеної розрядності?
5. Як працює команда MUL?
6. Як запрограмувати цикли на асемблері при обмеженій кількості регістрів?

Лабораторна робота №5. Програмування побітових операцій

Мета: Навчитися програмувати на асемблері побітові операції, вивчити основні команди обробки бітів.

Завдання

1. Створити у середовищі Microsoft Visual Studio проект з ім'ям **Lab5**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main5.asm**. Цей модуль створити та написати заново;
   - другий модуль: використати **module** попередніх робіт;
   - третій модуль: модуль **longop** попередньої роботи №4 доповнити новим кодом відповідно завданню.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуваний файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати, вихідний текст та дизасемблерний машинний код програми.

Теоретичні відомості

Програмісту на асемблері потрібно враховувати, що внутрішнє представлення інформації та її обробка, зокрема, процесором, грунтується на двійкових кодах. Кожна команда так чи інакше змінює біти у регістрах або пам'яті. Проте можна виділити деякі команди, які прийнято називати командами "побітових", або порозрядних операцій. Вони надають програмісту широкі можливості маніпулювати окремими бітами або групами бітів даних.

Важливим також є те, що побітові команди відносяться до найшвидших команд для усіх цифрових процесорів. Кожний хто програмує на асемблері, повинен знати та вміти використовувати такі команди. Це також сприяє розробці та реалізації ефективних алгоритмів та програм.

Далі розглянемо основні групи таких команд

Команди побітових операцій

Команда AND. Побітова кон'юнкція двійкових кодів двох операндів

```
and dest, src
```

Приклад

```
mov al, 75h            ; 01110101
and al, 3Eh            ; 00111110
```

У деяких алгоритмах побітову кон'юнкцію використовують для виділення, "вирізання" окремих бітів. Для виділення окремого біту виконується побітове AND з відповідною "маскою" – двійковим кодом у якому потрібний біт дорівнює 1, а усі решта бітів – 0, наприклад:

```
and eax, 00004000h     ; "вирізання" 14-го біту маскою 0..0 0100 0000 0000 0000
and cx, 0008h          ; "вирізання" 3-го біту маскою 0..0 1000
and edx, 0F0000000h    ; "вирізання" чотирьох старших бітів у регістрі EDX
and ebx, 0FFFF7FFFh    ; обнулення  15-го біту у регістрі EBX маскою 1..101..1
```

Після того, як виконано операцію AND, наприклад, EAX із операндом-маскою **00004000h**, то, щоб знайти чому дорівнює окремий 14-й біт регістру EAX, порівнюємо EAX з нулем, наприклад, командою CMP:

```
and eax, 00004000h
cmp eax, 0
```

Якщо EAX дорівнює 0, то досліджуваний 14-біт є нульовим.

Команда OR. Побітова диз'юнкція двійкових кодів двох операндів

```
or dest, src
```

Приклад

```
mov al, 60h            ; 01100000
or al, 3Ah             ; 00111010
                 ; AL = 01111010
or eax, 00008000h     ; 15-й біт регістру EAX стає 1, решта бітів не змінюється
```

Команда XOR. Нерівнозначність (*eXclusive OR*) бітів двійкових кодів двох операндів.

```
xor dest, src
```

Якщо біт першого операнду дорівнює відповідному біту другого операнду, то біт результату (операнду dest) буде 0, інакше 1, наприклад:

```
mov al, 75h          ; 01110101
xor al, 3Eh          ; 00111110
                ;AL = 01001011

mov al, 75h          ; 01110101
xor al, 20h          ; 00100000
                ;AL = 01010101   інверсія 5-го біту

mov al, 75h          ; 01110101
xor al, 0FFh         ; 11111111
                ;AL = 10001010   інверсія усіх бітів регітру AL
```

Дуже популярним прийомом є використання команди XOR для обнулення регістрів, наприклад:

```
xor eax, eax        ; EAX = 0
```

Це працює швидше, аніж

```
mov eax, 0
```

Команда NOT. Виконується побітова інверсія – нульові біти замінюються на 1, а одиничні біти стають 0.

Приклад

```
mov al, 75h         ; 01110101
not al              ; 10001010
```

Команди зсуву

Такі команди зсувають біти двійкового коду операндів.

Команда SHR. Зсув бітів вправо (у напрямку молодшого біту).

```
shr dest, count
```

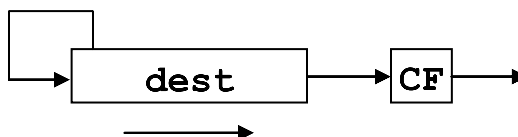
Виконується зсув бітів коду, записаного у операнді **`dest`**, вправо на **`count`** бітів. У старші **`count`** бітів записуються нулі.

Операнд **`dest`** може вказувати регістр або адресу пам'яті. Операнд **`count`** може бути безпосереднім значенням або регістром CL. Значення операнду **`count`** у 32-бітовому режимі процесора може бути від 0 до 31.

Приклад

```
mov eax, 0F00B000h     ;00001111000000001011000000000000
shr eax, 7             ;00000000000111100000000101100000
```

Команда SAR. Зсув бітів вправо арифметичний. Старший (лівий) біт розмножується.



Якщо лівий біт операнду **`dest`** дорівнює 0, то зсув виконується так само, як і для команди SHR – зліва записуються нулі, наприклад

```
mov eax, 0F00B000h     ;00001111000000001011000000000000
sar eax, 7             ;00000000000111100000000101100000
```

А якщо старший біт операнду є 1, то при зсуві розмножуються вже одиниці, наприклад

```
mov eax, -1580246784   ;10100001110011110101100100000000
sar eax, 7             ;11111111010000111001111010110010
```

Команди SAL, SHL. Зсув бітів вліво (у напрямку старшого біту).

```
shl dest, count
```

Виконується зсув бітів коду, записаного у операнді **dest**, вліво на **count** бітів. У молодші **count** бітів записуються нулі.

Операнд **dest** може вказувати регістр або адресу пам'яті. Операнд **count** може бути безпосереднім значенням або регістром CL. Значення операнду **count** у 32-бітовому режимі процесора може бути від 0 до 31.

Приклад

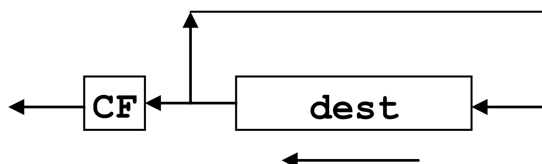
```
mov ax, 2Fh    ;0000 0000 0010 1111
mov cl, 7      ;визначаємо зсув на 7 бітів
shl ax, cl     ;0001 0111 1000 0000

mov eax, 2Fh   ;0000 0000 0000 0000 0000 0000 0010 1111
shl eax, 13    ;0000 0000 0000 0101 1110 0000 0000 0000

mov eax, 2Fh   ;0000 0000 0000 0000 0000 0000 0010 1111
sal eax, 13    ;0000 0000 0000 0101 1110 0000 0000 0000
```

Примітка. Команди SAL та SHL працюють однаково.

Команди циклічного зсуву

Команда ROL. Циклічний зсув бітів вліво (у напрямку старшого біту). Старші біти записуються на молодші позиції.

```
rol dest, count
```

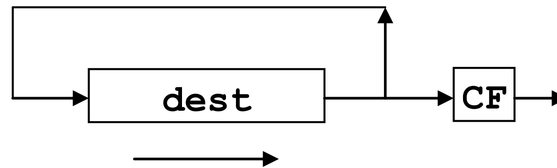


Приклад

```
mov ax, 7215h       ;0111 0010 0001 0101
rol ax, 4           ;0010 0001 0101 0111
```

Команда ROR. Циклічний зсув бітів вправо (у напрямку молодшого біту). Старші біти записуються на молодші позиції.

```
ror dest, count
```



Приклад

```
mov ax, 7215h        ;0111 0010 0001 0101
ror ax, 4            ;0101 0111 0010 0001
```

Приклади доступу до окремих бітів даних

Як узнати значення потрібного біту, наприклад, 67-го біту у 128-бітовому блоці даних? Вирішити це завдання можна наступним чином. Розглянемо блок даних як масив байтів (рис. 5.1).

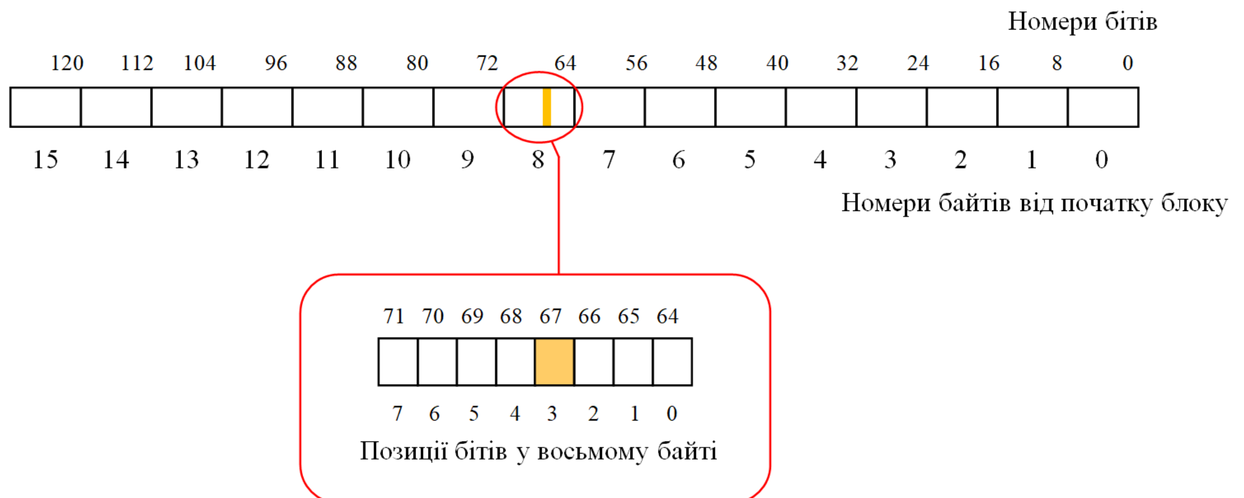


Рис. 5.1. Блок даних як масив байтів

У якому байті знаходиться потрібний біт? Номер байту дорівнює номеру біту, діленому на вісім: 67/8 = 8. Таким чином, 67-й біт знаходиться у 8-му байті.

Позиція потрібного біту у байті є залишком від ділення номера біту на 8. Для 67-го біту це позиція 3.

Усе це запрограмувати можна, наприклад, так:

```
mov al, byte ptr [myData+8]      ; у регістр AL завантажуємо 8-й байт
and al, 08h                      ; "вирізаємо" 3-й біт:
                                 ;            AL = XXXX XXXX
                                 ; and       08h = 0000 1000
                                 ;                 ---------
                                 ; результат: AL = 0000 X000
```

А як запрограмувати читання будь-якого довільного N-го біту? Для цього потрібно обчислювати номер байту та позицію біту. Ділення N на 8 та знаходження залишку від ділення можна виконати одною командою DIV, проте вона працює не швидко і є не зовсім зручною для вирішення нашого завдання.

Ділення на 8 – це те саме, що зсув на 3 біти вправо. Так можна запрограмувати обчислення номеру байту.

Знаходження залишку від ділення числа на 8 можна запрограмувати як вирізання трьох молодших бітів цього числа. Після знаходження залишку потрібно сформувати бітову маску – цю маску можна уявити як результат зсуву коду 00000001b вліво відповідно позиції потрібного біту.

Приклад реалізації наведеного вище алгоритму

```
mov ebx, Nbit                ; Nbit – це номер біту
mov ecx, ebx
shr ebx, 3                   ; номер байту

and ecx, 07h                 ; бітова позиція = вирізаємо 3 молодші біти
mov al, 1
shl al, cl                   ; AL = маска вирізання біту Nbit

mov ah, byte ptr [myData+ebx]
and ah, al                   ; результат у регістрі AH
```

Значення Nbit-го біту міститься у регістрі AH, про що можна дізнатися так: якщо AH=0, то біт був нульовим, а якщо AH $\neq$ 0, то біт був 1.

Запис потрібного біту у блок даних. Спочатку так само можна знайти номер байту у блоці та бітову позицію у окремому байті. Далі, якщо треба встановити потрібний біт у 1, то виконується команда OR з 0..010..0. Якщо треба обнулити потрібний біт, то це можна зробити командою AND з 1..101..1, наприклад

```
  mov ah, 1               ; вказування потрібного значення біту

  mov ebx, Nbit           ; Nbit – це номер біту
  mov ecx, ebx
  shr ebx, 3              ; номер байту
  and ecx, 07h            ; позиція потрібного біту у байті
  mov al, 1
  shl al, cl              ; маска 0..010..0 за умовчанням

  cmp ah, 0
  jz @set0
  or byte ptr [myData+ebx], al
  jmp @goon

@set0:
  not al                  ; маска 1..101..1 для AND
  and byte ptr [myData+ebx], al

@goon:                    ; щось робимо далі
```

У цьому програмному коді потрібне значення Nbit-го біту від початку вказується у регістрі AH наступним чином: якщо AH=0, то Nbit-й біт блоку даних буде обнулюватися, а якщо AH $\neq$ 0, то цей біт стане 1. Решта бітів блоку даних не змінюватиметься.

Передача числових значень та вказівників у параметрах процедур

Нехай потрібно передати у процедуру числове значення. Процедура повинна його сприйняти, щось виконати, та записати результат у потрібне місце пам'яті (наприклад, у якусь перемінну). Таким чином, у процедури можуть бути два параметри: один з них вона буде сприймати як просте числове значення, а другий параметр – як вказівник результату. По аналогії з мовою C/C++

```
void DoSomething(long *dest, long value);
```

Приклад на асемблері

```
.data
  varA dd 2014
  varB dd ?
. . .

.code
```

```
; ця процедура має два параметри, які передаються через стек
DoSomething proc
  push ebp              ;пролог
  mov ebp, esp

  mov eax, [ebp+12]     ;перший параметр - звичайне число
  add eax, 8            ;якось його використовуємо

  mov ebx, [ebp+8]      ;другий параметр - вказівник
  mov [ebx], eax        ;запис значення EAX у пам'ять по вказівнику

  mov esp, ebp          ;епілог процедури - відновлюємо стек
  pop ebp
  ret 8                 ;у стеку були два параметри - 8 байтів
DoSomething endp
. . .

  push varA             ;перший параметр - значення перемінної varA
  push offset varB      ;другий параметр - адреса перемінної varB
  call DoSomething
```

Процедура DoSomething запише у перемінну varB значення 2022.

Для того, щоб у тілі процедури записати якійсь результат по адресі, на яку вказує параметр-вказівник, потрібно значення цього параметру спочатку завантажити у регістр, наприклад, EBX. А потім вже так:

```
mov [ebx], eax           ;запис значення EAX у пам'ять по вказівнику
```

або те саме, записане коректніше:

```
mov dword ptr[ebx], eax
```

Можливий універсальний підхід до вирішення завдань

У даній лабораторній роботі пропонуються завдання запрограмувати виконання якихось побітових операції над бітовими даними (масивами) розрядністю до сотень бітів.

Безумовно, кожне завдання може вирішуватися якимось унікальним способом, який може бути специфічним для кожного варіанту завдання. Часто буває, що рішення є достатньо складним. Як саме вирішувати завдання – студент обирає індивідуально.

Але, можна запропонувати для вирішення багатьох (а, можливо, і для усіх) варіантів завдань деяку універсальну схему. У різних варіаціях цю схему можна використати для кожного варіанту завдань, причому можна

відзначити достатню простоту та зрозумілість підходу, який пропонується. У якості прикладу можна розглянути зсув бітів на деяку кількість позицій.

Універсальна схема, представлена у вигляді С-подібного псевдокоду, може бути такою:

```
posSrc = стартова позиція біта звідки
posDest = стартова позиція біта куди
масив-копія = первісний масив        //копіювання, якщо потрібно
обнулення масиву результату          //обнулення, якщо потрібно
while (по усім бітам багаторозрядного масиву)
  {
  біт = ReadOneBit(posSrc)        //читаємо з масиву-копії
  WriteOneBit(біт)
  posSrc ++
  posDest ++
  }
```

Процедура **ReadOneBit** читає один біт, а процедура **WriteOneBit** – записує один біт так, як розглянуто вище у пп. Приклади доступу до окремих бітів даних.

Загальний підхід можна сформулювати так: у циклі читається по одному біту з одної позиції і записується цей біт (можливо, якось трансформований) у іншу позицію бітового поля.

Що конкретно виконується – визначається стартовими позиціями для бітів читання та запису. Наприклад, якщо **posSrc** = позиція старшого біта, а **posDest** = **posSrc** – 1, то фактично виконується зсув бітів праворуч (від молодших до старших), а якщо posSrc = 0, а posDest = M, то виконується зсув вліво на M бітів. І тому подібне.

Методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім’ям **Lab5**.

2. Додайте у проект порожній файл з ім’ям **main5.asm**. Цей файл буде головним файлом програмного коду.

3. Додайте у проект модулі **module** та **longop**. У проекті використовується файли **module.asm, module.inc, longop.asm**, **longop.inc** попередньої роботи №4.

4. Запрограмуйте у модулі **longop** процедуру обробки даних підвищеної розрядності згідно варіанту завдання. Назвіть процедуру ім’ям, наприклад, **Shr**. Рекомендується надати таке ім’я, яке позначає не тільки функцію, а й

приналежність модулю, у якому ця процедура міститься, наприклад, **Shr_LONGOP**. Це дозволяє запобігати конфліктів імен у складних багатомодульних проектах.

5. У файлі **main5.asm** потрібно запрограмувати виклик процедури обробки даних підвищеної розрядності з тестовими значеннями параметрів та вивід результатів у шістнадцятковій системі числення у діалоговому вікні MessageBox. Запрограмувати вивід вихідних даних та результатів обробки у шістнадцятковому коді.

6. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio. Відомості та методичні рекомендації надані у відповідних розділах попередніх робіт.

Варіанти завдань та основні вимоги

Потрібно запрограмувати процедуру, яка обробляє дані підвищеної розрядності. У процедури мають бути такі параметри: адреса джерела даних, адреса результату, розрядність, а також (якщо потрібні) параметри N, M. Параметри N та M повинні бути довільними цілими. Якщо студент обмежить їх, наприклад, тільки кратними 8, то оцінка буде зменшуватися.

Перелік варіантів наданий таблиці

Таблиця 5.1

№ вар.	Операція	Розрядність (біт)
1	Зсув бітів вправо (подібно SHR) на N розрядів	256
2	Зсув бітів вправо арифметичний (подібно SAR) на N розрядів	288
3	Зсув бітів вліво (подібно SHL) на N розрядів	320
4	Зсув бітів вліво циклічний (подібно ROL) на N розрядів	352
5	Зсув бітів вправо циклічний (подібно ROR) на N розрядів	384
6	Обчислення кількості одиниць у двійковому коді	416
7	Обчислення кількості нулів у двійковому коді	448
8	Обчислення кількості старших нулів у двійковому коді	480
9	Обчислення кількості старших одиниць у двійковому коді	512
10	Запис M нулів починаючи з N-го біту. Решту бітів зробити 1	544
11	Запис M одиниць починаючи з N-го біту. Решту бітів зробити 0	576
12	Зсув вліво, щоб старший біт був 1 (нормалізація). Після зсуву у регістр EAX записати, на скільки розрядів був виконаний зсув	608
13	Зсув M старших бітів на N позицій вліво. Решта бітів нерухомі	640
14	Зсув M молодших бітів на N позицій вправо. Решта бітів нерухомі	672
15	Оберт бітів дзеркально відносно середньої позиції коду	704
16	Зсув вліво на N розрядів. У молодші N розрядів записується	736

	двійковий код, який вказується відповідним параметром процедури	
17	Зсув вправо на N розрядів. У старші N розрядів записується двійковий код, який вказується відповідним параметром процедури	768
18	Починаючи з N-го розряду виконується побітове AND з M-бітовою маскою, яка вказується відповідним параметром процедури	800
19	Починаючи з N-го розряду виконується побітове OR з M-бітовою маскою, яка вказується відповідним параметром процедури	832
20	Починаючи з N-го розряду виконується побітове XOR з M-бітовою маскою, яка вказується відповідним параметром процедури	864
21	Починаючи з N-го розряду виконується інверсія M бітів	896
22	Запис M бітів, починаючи з N-го розряду	928
23	З джерела даних, починаючи з N-го біту, взяти M бітів (M≤32) і записати у регістр EAX	960
24	Обнулити M бітів, починаючи з N-го розряду	992
25	Запис 1 у M бітів, починаючи з N-го розряду	1024
26	Запис M одиниць починаючи з N-го біту. Решту бітів зробити 0	1056
27	Зсув бітів вліво циклічний (подібно ROL) на N розрядів	1088
28	Зсув бітів вправо циклічний (подібно ROR) на N розрядів	1120
29	Зсув бітів вправо арифметичний (подібно SAR) на N розрядів	1152
30	Обчислення кількості одиниць у двійковому коді	1184

Примітка. Розрядності операндів та інші параметри варіантів завдань можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасемблерного машинного коду
6. Висновки

Контрольні запитання

1. Що таке побітові операції?
2. Чим відрізняється команда SHR від SAL?
3. Який результат команди XOR і як вона може бути використана?
4. Яка команда виконує інверсію бітів?
5. Як можна узнати значення окремого біту?
6. Як можна записати кудись окремий біт?
7. Як зберігати та обробляти дані підвищеної розрядності?
8. Як запрограмувати параметри процедури?
9. Як запрограмувати запис результату через параметр процедури?
10. Як передати процедурі для обробки дані підвищеної розрядності?

Лабораторна робота №6. Програмування операцій ділення чисел

Мета: Навчитися програмувати на асемблері ділення чисел та перетворення з двійкової у десяткову систему числення.

Завдання

1. Створити у середовищі MS Visual Studio проект з ім'ям **Lab6**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main6.asm**. Цей модуль створити та написати заново, частково використавши текст модуля main5.asm попередньої роботи №5;
   - другий модуль: модуль **module** попередньої роботи №5;
   - третій модуль: модуль **longop** попередньої роботи №5.
3. Додати у модулі процедури, які потрібні для виконання завдання. Обгрунтувати розподіл процедур по модулям.
4. У цьому проекті кожний модуль може окремо компілюватися.
5. Скомпілювати вихідний текст і отримати виконуваний файл програми.
6. Перевірити роботу програми. Налагодити програму.
7. Отримати результати – кодовані значення чисел згідно варіанту завдання.
8. Проаналізувати та прокоментувати результати та вихідний текст.

Теоретичні відомості

Цілочисельне ділення

Нехай дані цілі числа *A* та *B*. Результатом ділення *A/B* може бути ціле або дробове число. Цілочисельним діленням є представлення результату ділення у вигляді двох цілих чисел: часткового *D* та залишку *R*. Запишемо це наступним чином:

$$\frac{A}{B} = D + \frac{R}{B} \qquad (6.1)$$

де: *D* – ціла частка результату, *R/B* – дробова частка результату.

Значення залишку $R = A \bmod B$, тобто *R* може бути від 0 до *B*-1.

Яка розрядність потрібна для представлення результатів ділення? Будемо розглядати ділення у двійковій системі числення. Введемо позначення: n_A, n_B, n_D, n_R – розрядність (кількість бітів) відповідно для *A, B, D* та *R*.

Якщо ділене *A* представлене n_A – бітовим двійковим кодом, то для часткового (*D*) потрібно бітів від (n_A - n_B +1) до n_A у залежності від значення дільника *B*. Наприклад, якщо n_A = 16, то максимальне значення для *A*

***A* = 1111111111111111**

Якщо n_B = 16, то у випадку максимального 16-бітового значення *B*

B = 1111111111111111

результатом ділення

$$\frac{A}{B} = \frac{1111111111111111}{1111111111111111} = 1$$

буде значення *D* = 1, для представлення якого достатньо одного біту – іншими словами: $n_D = n_A - n_B + 1 = 16 – 16 + 1 = 1$.

Якщо у двійковому коді *B* буде декілька, наприклад, п’ять старших нулів:

$$\frac{A}{B} = \frac{1111111111111111}{0000011111111111} = 100000\ (D),\text{ залишок } R = 11111$$

то, фактично, ділення не на 16-бітове, а на 11-бітове число, тому можна вважати n_B = 11. Результат (*D*) є 6-бітовим числом: $n_D = n_A - n_B +1 = 16 – 11 + 1 = 6$.

Ще приклад:

$$\frac{A}{B} = \frac{1111111111111111}{0000010000000000} = 111111\ (D),\text{ залишок } R = 1111111111$$

Загалом, для представлення залишку (*R*) достатньо $n_R = n_B$ розрядів, оскільки *R* не може бути більшим, аніж *B*-1.

У випадку мінімального ненульового значення *B*=1

$$\frac{A}{B} = \frac{1111111111111111}{0000000000000001} = 1111111111111111$$

результат ділення буде *D*=*A* із відповідною розрядністю $n_D = n_A$.

Алгоритм ділення цілих чисел "у стовпчик"

Розглянемо виконання операції $D = A/B$, де A та B цілі двійкові числа без знаку. Одним з найпростіших алгоритмів є ділення "у стовпчик". Нехай A=110101110101 та B=1010. Алгоритм ділення n-бітового числа A на 4-бітове B можна записати так:

1. Встановити i=n-4.
2. Взяти чотири старші біти числа A.
   Позначити це як $R=\{a_{n-1}, a_{n-2}, a_{n-3}, a_{n-4}\}$
3. Якщо R більше, або дорівнює B,
   то: i-та цифра (біт) результату d_i дорівнює 1, віднімаємо $R = R - B$
   інакше: i-та цифра результату d_i дорівнює 0
4. Зменшити i на одиницю. Якщо i менше нуля, то кінець роботи.
5. Помножити R на два. Таке множення означає зсув бітів на одну позицію вліво. Додати до R у молодший розряд i-й біт числа A, тобто цифру a_i.
6. Перехід на п. 2.

У результаті роботи алгоритму отримаємо біти D та залишку R (рис. 6.1)

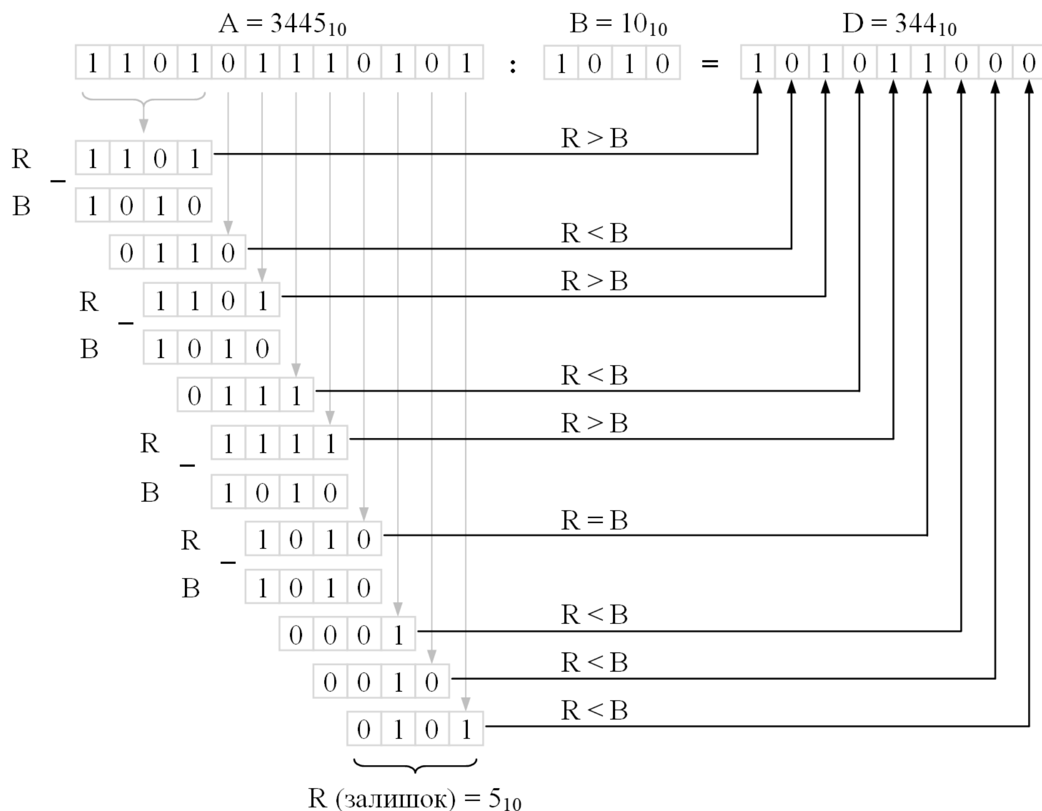


Рис. 6.1. Приклад ділення двійкових чисел ”у стовпчик”

Алгоритм ділення групами бітів

Якщо у системі команд процесора є команди ділення, то можна побудувати алгоритм ділення підвищеної розрядності на основі послідовності таких команд. Різновид алгоритму ділення D = A/B, який розглядається нижче, ґрунтується на тому, що багаторозрядне ділиме (A) у позиційній системі числення – у тому числі й у двійковому коді, може бути записане у вигляді суми груп розрядів. Наприклад, 128-бітове число A можна записати як суму чотирьох 32-бітових груп

$$A = 2^{96}A_3 + 2^{64}A_2 + 2^{32}A_1 + A_0 \qquad (6.2)$$

Розглянемо алгоритм ділення підвищеної розрядності (N) групами по бітів. Для операції D = A/B розрядність ділимого (A) нехай буде N=128, а дільник (B) 32-бітовий. Число A складається з чотирьох 32-бітових груп розрядів. Можна записати, що

$$D = \frac{2^{96}A_3 + 2^{64}A_2 + 2^{32}A_1 + A_0}{B} = 2^{96}\frac{A_3}{B} + 2^{64}\frac{A_2}{B} + 2^{32}\frac{A_1}{B} + \frac{A_0}{B} \qquad (6.3)$$

Результат ділення 32-бітових A_3 на B можна записати у вигляді

$$\frac{A_3}{B} = D_3 + \frac{R_3}{B}$$

де: D_3 – часткове, R_3 – залишок. Перепишемо вираження (6.3) наступним чином

$$D = 2^{96}\left(D_3 + \frac{R_3}{B}\right) + 2^{64}\frac{A_2}{B} + 2^{32}\frac{A_1}{B} + \frac{A_0}{B}$$

або

$$D = 2^{96} D_3 + 2^{64}\frac{2^{32}R_3 + A_2}{B} + 2^{32}\frac{A_1}{B} + \frac{A_0}{B} \qquad (6.4)$$

Якщо 64-бітове $(2^{32}R_3 + A_2)$ поділити на B, то буде

$$\frac{2^{32}R_3 + A_2}{B} = D_2 + \frac{R_2}{B}$$

Підставимо це у вираження (6.4) і об'єднаємо R_2 з A_1

$$D = 2^{96} D_3 + 2^{64} D_2 + 2^{32}\frac{2^{32}R_2 + A_1}{B} + \frac{A_0}{B} \qquad (6.5)$$

Наступний крок: 64-бітове ($2^{32}R_2 + A_1$) поділимо на B

$$\frac{2^{32}R_2 + A_1}{B} = D_1 + \frac{R_1}{B}$$

Тоды вираження (6.5) можна переписати у таке

$$D = 2^{96} D_3 + 2^{64} D_2 + 2^{32} D_1 + \frac{2^{32}R_1 + A_0}{B}$$

Залишилося виконати ділення 64-бітового ($2^{32}R_1 + A_0$) на B

$$D = 2^{96} D_3 + 2^{64} D_2 + 2^{32} D_1 + D_0 + \frac{R_0}{B}$$

Кінцевий результат такого ділення записується 32-бітовими групами D_i на відповідних позиціях розрядної сітки, а також 32-бітовим залишком R_0. Процес ділення групами бітів можна відобразити наступним чином

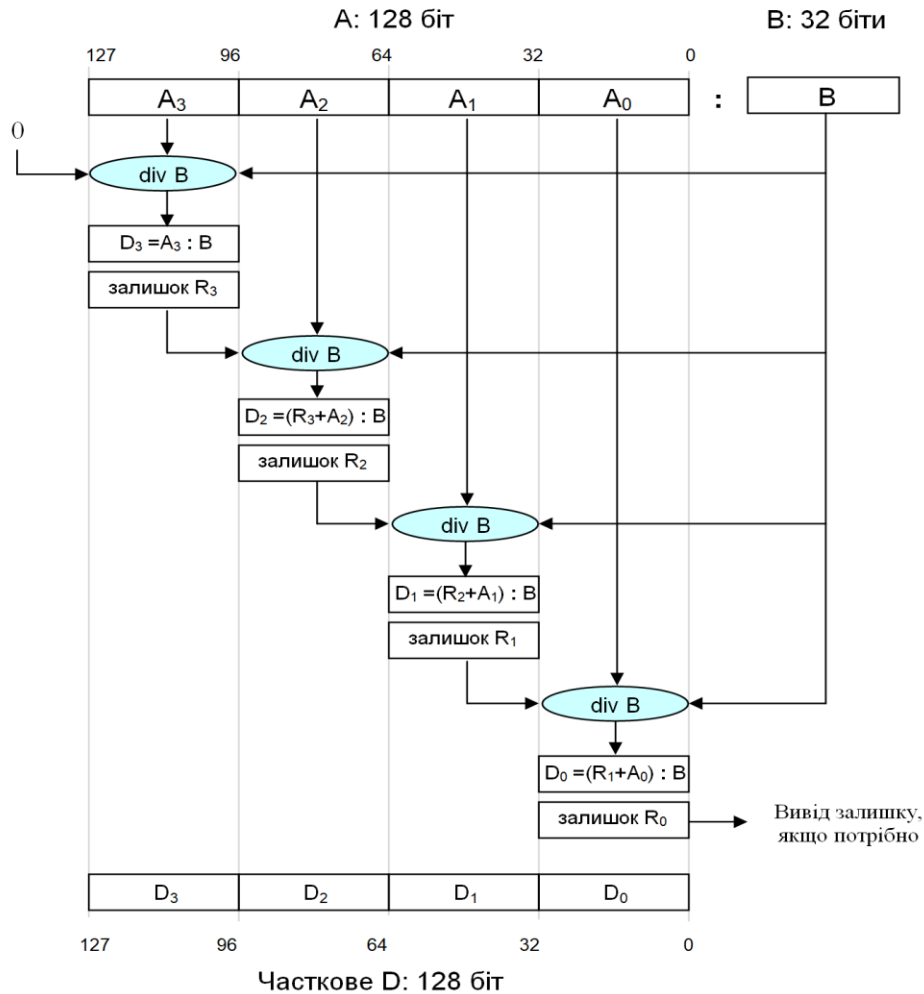


Рис. 6.2. Приклад ділення групами бітів "128:32"

Таким чином, операцію ділення 128-бітового A на 32-бітове B можна виконати чотирма кроками 32-бітного ділення A_i на B. На кожному кроці виконується цілочисельне ділення (DIV) двійкового 64-бітового числа на 32-бітове число. Результат кожного кроку у вигляді 32-бітового часткового D_i та 32-бітового залишку R_i . Залишок R_i на наступному кроці виступає старшою 32-бітовою частиною 64-бітового значення ($2^{32}R_i + A_{i-1}$).

Як ви вважаєте, чому розробники процесорів Intel спроектували команду DIV так, щоб залишок (R) від ділення 64-бітового значення регістрової пари EDX:EAX записувалися би саме у EDX, а результат ділення (D) – у регістр EAX?

Команди процесорів x86 для цілочисельного ділення

Команда DIV. Ця команда виконує ділення цілих чисел без знаку: A/B. Синтаксис запису:

```
div операнд
```

Для цієї команди записується тільки один операнд – дільник (*B*), який може бути 8-, 16-, 32- або 64-бітовим. Виконання команди DIV визначається розрядністю цього операнду. У залежності від розрядності операнду *B*, у якості діленого (*A*) буде: або вміст регістру AX, або вміст пари регістрів DX:AX, або вміст пари регістрів EDX:EAX, або вміст пари регістрів RDX:RAX.

Таблиця 6.1

Розміри операндів A/B	Неявний операнд (ділене A)	Операнд команди (дільник B)	Результати виконання команди DIV		
			Часткове	Залишок	Макс. значення часткового
16/8	AX	r/m8	AL	AH	2^8-1 = 255
32/16	DX:AX	r/m16	AX	DX	2^{16}-1 = 65535
64/32	EDX:EAX	r/m32	EAX	EDX	2^{32}-1
128/64	RDX:RAX	r/m64	RAX	RDX	2^{64}-1

Приклад. Поділити число *A*=60324 на *B*=2014. Дільник *B* буде 16-бітовим операндом команди DIV. Тоді для діленого потрібно використати пару регістрів DX:AX. Відповідний програмний код наданий нижче:

```
xor dx, dx        ;обнулюємо старші 16 бітів діленого
mov ax, 60324     ;молодші 16-бітів діленого
mov bx, 2014      ;16-бітовий дільник
div bx            ;ділення DX:AX / BX
```

Результат: часткове у регістрі AX = 29, залишок у регістрі DX = 1918.

При програмуванні ділення командою DIV необхідно враховувати:

- не можна ділити на нуль;
- обмеження максимальних значень результату, наведені вище у таблиці.

У цих випадках буде згенероване відповідне виключення (*exception*) і програма аварійно завершиться. Наступний приклад:

```
mov dx, 1          ;старші 16 бітів діленого
mov ax, 0FFFFh     ;молодші 16 бітів діленого
mov bx, 1          ;16-бітовий дільник
div bx ;спроба ділення DX:AX / BX = 1FFFF / 1 = 1FFFF ?
```

При виконанні команди DIV тут виникне помилка переповнення, оскільки результат 1FFFF не є 16-бітовим.

На відміну цього, наступний код виконується коректно:

```
xor edx, edx        ;обнулюємо старші 32 бітів діленого
mov eax, 1FFFFh     ;молодші 32 бітів діленого
mov ebx, 1          ;32-бітовий дільник
div ebx     ;ділення EDX:EAX / EBX = 1FFFF / 1 = 1FFFF
```

Результати такого ділення:

- часткове (1FFFF) записується у регістр EAX;
- залишок (0) записується у регістр EDX.

Команда IDIV. Ця команда виконує ділення цілих чисел зі знаком: A/B. Синтаксис запису:

```
idiv операнд
```

Так само, як і для команди DIV, для команди IDIV явно вказується тільки один операнд – дільник (*B*). Виконання команди IDIV визначається розрядністю цього операнду (табл.6.2).

Таблиця 6.2

Розміри операндів A/B	Неявний операнд (ділене A)	Операнд команди (дільник B)	Результати виконання команди IDIV		
			Часткове	Залишок	Діапазон значень часткового
16/8	AX	r/m8	AL	AH	-128 . . . +127
32/16	DX:AX	r/m16	AX	DX	-32768 . . . +32768
64/32	EDX:EAX	r/m32	EAX	EDX	-2^{31} . . . $2^{31}-1$
128/64	RDX:RAX	r/m64	RAX	RDX	-2^{63} . . . $2^{63}-1$

Якщо результат ділення від'ємний, то часткове та залишок записуються як від'ємні числа у додатковому коді відповідної розрядності.

Приклад. Поділити число *A*= -100000 на *B*=2014. Для представлення числа *A* потрібно як мінімум 32-бітів додаткового коду: *A* = FFFE7960. Число *B* може бути представлене 16-бітовим кодом. Можна вказати, що взагалі не варто писати програмний код для ділення констант, проте для ілюстрації роботи команди IDIV можна записати наступний програмний код:

```
mov dx, 0FFFEh    ;старші 16 бітів діленого A
mov ax, 7960h     ;молодші 16 бітів A
mov cx, 2014      ;дільник B
idiv cx        ;ділення DX:AX / CX = FFFE 7960 / 2014
```

У результаті виконання команди IDIV
- часткове FFCF (-49) записується у регістр AX;
- залишок FADE (-1314) записується у регістр DX.

Можна поставити запитання: як запрограмувати ділення перемінних, наданих у 32-бітовому форматі? Це можна зробити, наприклад, так, як наведено нижче

```
.data
varA dd -100000                ;FFFE7960
varB dd 2014                   ;000007DE

.code
mov eax, dword ptr [varA]      ;EAX = FFFE7960
cdq                            ;EDX = FFFFFFFF
mov ecx, dword ptr [varB]      ;ECX = 000007DE
idiv ecx                       ;ділення EDX:EAX / ECX
```

У цьому коді використано команду CDQ, яка перетворює подвійне слово зі знаком, записане у регістрі EAX, у квадрослово EDX:EAX. Значення старшого (знакового) біту регістру EAX розмножується і записується у регістрі EDX. Для від'ємного числа у регістрі EDX усі 32 біти будуть 1.

У результати виконання ділення для наведеного вище коду:

- часткове FFFFFFCF (-49) записується у регістр EAX;
- залишок FFFFFADE (-1314) записується у регістр EDX.

Команди зсуву та їхнє використання у арифметичних операціях

Такі команди зсувають біти двійкового коду операндів. Вони вже були розглянуті у теоретичних відомостях Їх ми вже розглядали у попередній лабораторній роботі №6. Але тут ми їх використаємо щоб порівняти з діленням та множенням.

Команда SHR. Зсув бітів вправо (у напрямку молодшого біту).

```
shr dest, count
```

Розглянемо приклад зсуву

```
mov eax, 0F00B000h ;00001111000000001011000000000000 = 251703296
shr eax, 7         ;00000000000111100000000101100000 = 1966432
```

Зсув вправо на 7 бітів дає той самий результат, що й ділення без знаку на 2^7, тобто 1966432 = 251703296 / 128. Таким чином, якщо потрібно запрограмувати спрощене ділення цілих чисел без знаку на степінь 2, то замість команди DIV можна використовувати SHR, яка працює набагато швидше.

Команда SAR. Зсув бітів вправо арифметичний. Старший (лівий) біт розмножується, наприклад

```
mov eax, -1580246784   ;10100001110011110101100100000000
sar eax, 7             ;11111111010000111001111010110010
```

Команда SAR подібна діленню цілих зі знаком на степінь 2. У наведеному вище прикладі -1580246784 / 128 = -12345678. Виникає питання: якщо дільник степінь 2, то чи можна у програмах ділення використовувати SAR замість IDIV? Для порівняння можна розглянути код на основі команди IDIV

```
mov eax, -1580246784
cdq
mov ecx, 128
idiv ecx      ; результат: EAX = FF439EB2 (-12345678), EDX = 00000000
```

Тобто, команда IDIV записує у регістр EAX той самий результат, як і для наведеного вище коду для команди SAR.

Розглянемо інший приклад – ділення (-3/4)

```
mov eax, -3
cdq
mov ecx, 4
idiv ecx    ;результат: EAX=00000000, EDX=FFFFFFFD (-3)
```

Команда IDIV дає результат: (-3/4) = 0, залишок (-3).

А тепер запишемо ділення (-3/4) на основі команди SAR

```
mov eax, -3     ; FFFFFFFD
sar eax, 2      ; результат: EAX = FFFFFFFF (-1)
```

Команда SAR дає інший результат: (-3/4) = -1. На відміну від IDIV, після арифметичного зсуву SAR отримаємо результат, округлений до -1.

Таким чином, програмісту у кожному конкретному випадку треба ретельно вивчати, чи коректно заради прискорення замість ділення використовувати зсув.

Команди SAL, SHL. Зсув бітів вліво (у напрямку старшого біту).

```
shl dest, count
```

Виконується зсув бітів коду, записаного у операнді **dest**, вліво на **count** бітів. У молодші **count** бітів записуються нулі, наприклад

```
mov ax, 2Fh          ; 0000 0000 0010 1111
mov cl, 7
shl ax, cl           ; 0001 0111 1000 0000
```

Якщо розглядати результат зсуву 2Fh << 7 = 1780h, або у десятковій системі 47 << 7 = 6016, то зсув на 7 бітів – це те саме, що множення на 2^7. Тобто, $47 \times 2^7 = 47 \times 128 = 6016$.

Перетворення з двійкової у десяткову систему числення

Одним з відомих алгоритмів перетворення чисел у іншу позиційну систему числення є цілочисельне ділення на основу нової системи. Після кожного ділення береться залишок – він буде цифрою результату. Послідовне ділення продовжується доти результат ділення не стане нулем.

У якості прикладу двійкове число 110101110101 переведемо у десяткову систему послідовним діленням на 10

110101110101 : 1010 = 101011000 залишок 101 (десяткова цифра **5**)
101011000 : 1010 = 100010 залишок 100 (десяткова цифра **4**)
100010 : 1010 = 11 залишок 100 (десяткова цифра **4**)
11 : 1010 = 0 залишок 11 (десяткова цифра **3**)

Результат 3445_{10}

Вивід результатів у десятковій системі числення

Для того, щоб вивести якесь числове значення за допомогою процедури, яка відображає текст, потрібно сформувати відповідний рядок символів, який записати у масив – буфер рядка тексту. Потрібно обробити десяткові цифри числа, перетворивши кожну цифру у однобайтовий код ASCII. Для символів цифр 0, 1, 2, . . . , 9 відповідні коди ASCII будуть 48, 49, 50, . . . , 57. Таким чином, **код ASCII** = цифра + 48.

Це співвідношення можна використати при програмуванні циклу перетворення числового коду у рядок тексту з десятковими цифрами.

Методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім'ям **Lab6**.

2. Додайте у проект порожній файл з ім'ям **main6.asm**. Цей файл буде головним файлом програмного коду.

3. Додайте у проект модулі **module** та **longop**. У проекті використовується файли **module.asm, module.inc, longop.asm**, **longop.inc** попередньої роботи №5.

4. Запрограмуйте процедуру ділення чисел підвищеної розрядності на 10. Назвіть процедуру ім'ям, наприклад, **Div10**. Рекомендується надати таке ім'я, яке позначає не тільки функцію, а й приналежність модулю, у якому ця процедура міститься, наприклад, **Div10_LONGOP**. У процедури повинні бути такі аргументи: адреса числа підвищеної розрядності, розрядність (кількість бітів) та адреси результату – часткового та залишку від ділення на 10.

5. Запрограмуйте процедуру перетворення у десятковий код чисел підвищеної розрядності. Назвіть процедуру **StrDec**. У процедурі повинні бути три аргументи: адреса буфера тексту-результату, адреса числового значення, кількість бітів числа (параметри такі самі, як у процедури StrHex_MY). Для перетворення у десятковий код використати послідовне ділення на 10. Рекомендується для ділення на 10 викликати процедуру Div10, яка запрограмована у п. 4 даної роботи.

6. Потрібно вирішити, у яких модулях будуть розташовуватися програмні коди нових процедури Div10 та StrDec.

7. У файлі **main6.asm** потрібно запрограмувати цикл для обчислення значення факторіалу – рекомендується скористатися кодом попередньої роботи №4. Для виводу десяткового значення факторіалу треба викликати процедуру StrDec.

8. Також у файлі **main6.asm** потрібно запрограмувати обчислення значення виразу за формулою згідно варіанту завдання.

9. Запрограмувати вивід результатів у діалоговому вікні MessageBox. Запрограмувати вивід потрібних числових значень у десятковому коді.

10. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio подібно до того, як у попередніх лабораторних роботах.

Варіанти завдань та основні вимоги

1. Потрібно запрограмувати на асемблері вивід значення факторіалу n! у **десятковому коді**. Використати програмний код обчислення факторіалу лабораторної роботи №4. Для кожного студента своє значення n

$$n = 30 + 2\times H,$$

де H – це номер студента у журналі.

2. Перетворення у десятковий код запрограмувати діленням числа підвищеної розрядності на 10. Ділення на 10 виконати двома способами – діленням "у стовпчик" та діленням груп бітів. Ділення групами бітів запрограмувати або як 8-бітове, або як 16-бітове, або як 32-бітове відповідно до варіанту. Варіант ділення групами бітів обирається відповідно залишку ділення номеру студента у журналі (H) на 3:

- залишок = 0 – ділення групами по 8 бітів;
- залишок = 1 – ділення групами по 16 бітів;
- залишок = 2 – ділення групами по 32 бітів.

3. Програмний код ділення "у стовпчик" та ділення групами бітів оформити у вигляді процедур. Процедури повинні містити такі параметри: адреса ділимого, розрядність ділимого, значення дільника (B) та інші (за необхідності).

4. Запрограмувати на асемблері також обчислення формули, яку вибрати з таблиці, наведеної нижче. Номер варіанту формули згідно номеру в журналі. Узагальнено формула має вигляд $y = F(x, m)$.Значення x, y повинні бути 32-бітовими цілими зі знаком. Значення m – ціле без знаку. Результат (y) повинен записуватися у регістр EAX.

5. При програмування виражень для формул відповідно обраному варіанту проаналізувати можливість різної послідовності виконання операцій і обрати таку послідовність, яка забезпечує найвищу точність результату

Варіанти формул $y = F(x, m)$ наведені у таблиці 6.3 нижче

Таблиця 6.3

Варіанти формул

Номер	Формула	Номер	Формула	Номер	Формула
1	$y = \frac{x}{x+1} 2^{m}$	11	$y = \frac{x}{x+3} 2^{m+2}$	21	$y = \frac{x}{x+5} 2^{m+4}$
2	$y = \frac{x-1}{x+1} 2^{m}$	12	$y = \frac{x-3}{x+3} 2^{m-2}$	22	$y = \frac{x-5}{x+5} 2^{m-4}$
3	$y = \frac{x}{3} 2^{m}$	13	$y = \frac{x}{7} 2^{m+2}$	23	$y = \frac{x}{11} 2^{m+4}$
4	$y = \frac{3}{x+1} 2^{m}$	14	$y = \frac{7}{x+1} 2^{m}$	24	$y = \frac{11}{x+1} 2^{m}$
5	$y = \frac{x}{3} 2^{-m}$	15	$y = \frac{x}{7} 2^{-m}$	25	$y = \frac{x}{11} 2^{-m}$
6	$y = \frac{x}{x+2} 2^{m+1}$	16	$y = \frac{x}{x+4} 2^{m+3}$	26	$y = \frac{x}{x+6} 2^{m+5}$
7	$y = \frac{x-2}{x+2} 2^{m-1}$	17	$y = \frac{x-4}{x+4} 2^{m-3}$	27	$y = \frac{x-6}{x+6} 2^{m-5}$
8	$y = \frac{x}{5} 2^{m+1}$	18	$y = \frac{x}{9} 2^{m+3}$	28	$y = \frac{x}{13} 2^{m+5}$
9	$y = \frac{5}{x+1} 2^{m}$	19	$y = \frac{9}{x+1} 2^{m}$	29	$y = \frac{13}{x+1} 2^{m}$
10	$y = \frac{x}{5} 2^{-m}$	20	$y = \frac{x}{9} 2^{-m}$	30	$y = \frac{x}{13} 2^{-m}$

Примітка. Формули та інші параметри варіантів завдань можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасемблерного машинного коду
6. Висновки

Контрольні запитання

1. Що таке цілочисельне ділення?
2. Як виконується команда DIV?
3. Як виконується команда IDIV?
4. Коли можна замість ділення робити зсув?
5. Чим відрізняється команда SAR від SHR?
6. Як виконується ділення чисел підвищеної розрядності?
7. Як перевести з двійкової у десяткову систему?

Лабораторна робота №7. Виконання операцій з плаваючою точкою та вивчення команд x87 FPU

Мета: Навчитися програмувати операції з плаваючою точкою на асемблері.

Завдання

1. Створити у середовищі MS Visual Studio проект з ім’ям **Lab7**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути головний файл **main7.asm** та інші модулі (за необхідності).
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуваний файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – числові значення згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати та вихідний текст.

Теоретичні відомості

Середовище x87 FPU

Блок, який отримав назву x87 FPU (*Floating Point Unit*), виконує операції у форматах з плаваючою точкою. Спочатку це був окремий пристрій (сопроцесор 8087). Починаючи з Intel 80486 такий блок міститься усередині центрального процесора.

Для виконання у середовищі x87 FPU операцій з плаваючою точкою є декілька десятків команд, яки можна використовувати у програмах на асемблері. Для цього програмістам потрібно уявляти, хоча б загалом, з чого складається блок x87 FPU, які він має ресурси, та як їх можна використовувати. Іншими словами, програмісту надається деяке середовище – у даному випадку це x87 FPU, у якому виконуватимуться його програми. Це середовище утворюється з апаратних ресурсів, інтерфейсів доступу до них, та інших дозволів та заборон, які враховуються на певному програмному рівні.

Рис. 7.1. Елементи середовища x87 FPU

Регістр статусу. Цей 16-бітовий регістр зберігає інформацію про стан x87 FPU у вигляді значень бітів-прапорців

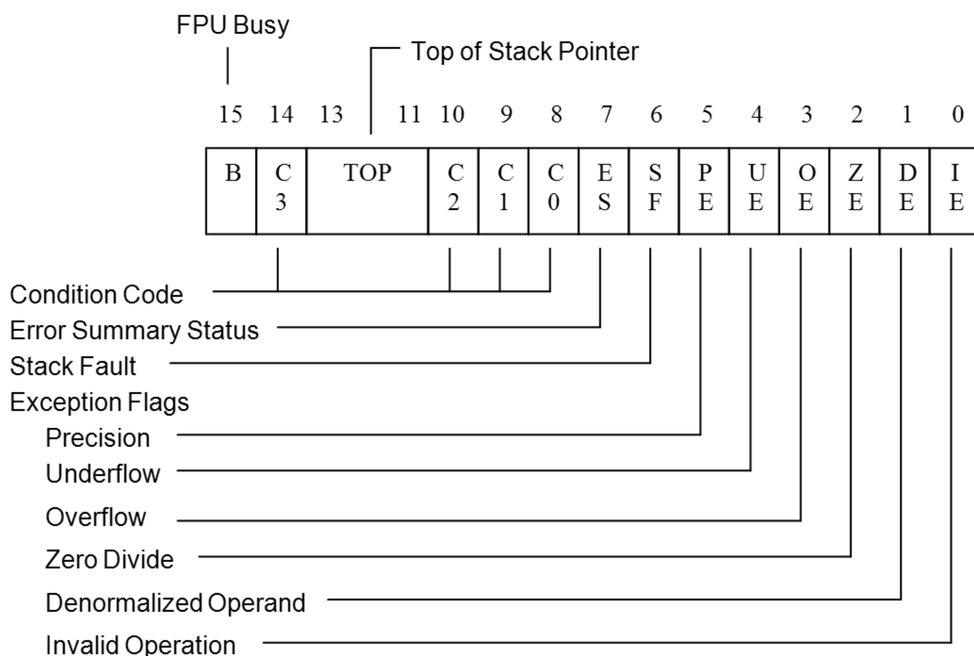


Рис.7.2. Регістр статусу x87 FPU (згідно джерела: Intel® 64 and IA-32 Architectures. Software Developer’s Manual).

Три біти 11-13 є значенням TOP (*Top of Stack Pointer*), яке означає адресу вершини ST(0) стеку даних. Іншими словами, TOP – це номер (від 0 до 7) регістру R_{TOP}, який на даний момент представляє вершину стеку.

Стек даних FPU

Для виконання операцій потрібно завантажувати дані у стек FPU.

Стек даних x87 FPU складається з восьми 80-бітових регістрів R0-R7. Цей стек організований як кільце. Доступ до стеку даних можливий тільки через його вершину, яка зветься ST(0). Вершині стеку відповідає один з регістрів R0-R7, на який вказує 3-х бітове значення вказівника TOP.

Запис до стеку. Це можна зробити командою FLD

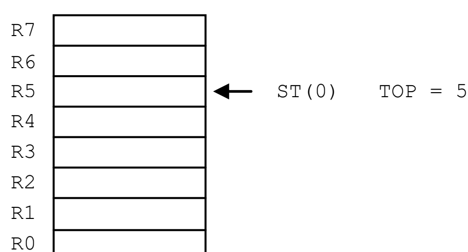
```
FLD src
```

Значення операнду src записується у вершину стека ST(0), наприклад

```
myvar dt 35.247
  . . .
fld tbyte ptr[myvar]
```

При виконанні команди FLD спочатку TOP зменшується на одиницю (причому, якщо TOP дорівнював 0, то буде TOP = 7) – так визначається нове розташування ST(0); а потім у відповідний регістр R_{TOP} записується числове значення операнду (рис. 7.3).

Стан стеку до FLD



Стан стеку після FLD

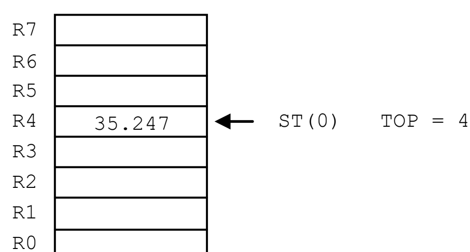


Рис. 7.3. Запис у стек одного числа командою FLD

Розглянемо приклад запису у стек x87 FPU декількох значень. Простежимо зміни стану стеку впродовж виконання команд FLD.

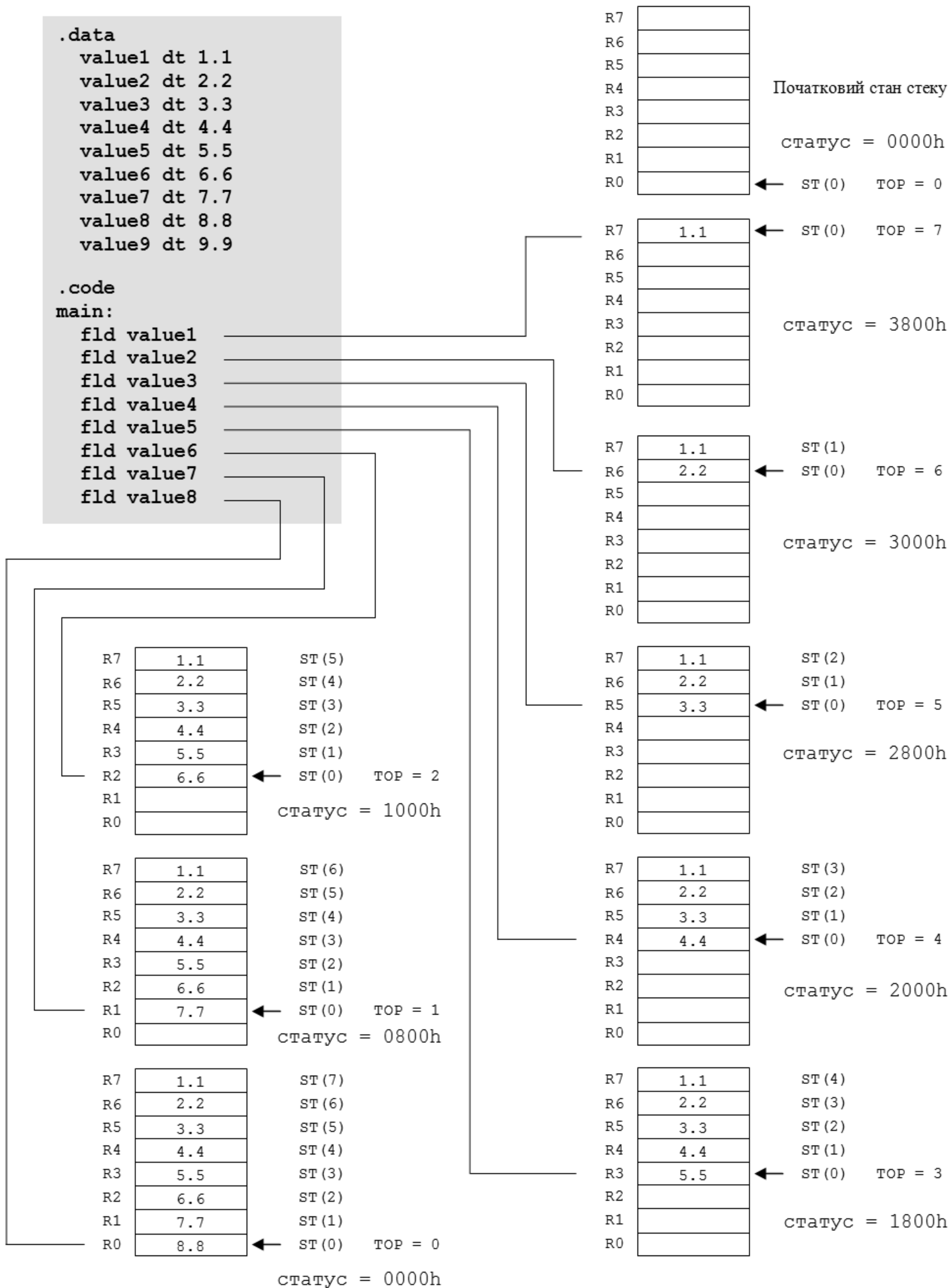


Рис. 7.4. Стек FPU може зберігати одночасно не більше восьми значень

Читання зі стеку. Це можна зробити командою FSTP

```
FSTP dest
```

Приклад:

```
myvar dt ?
. . .
fstp tbyte ptr[myvar]                    ; myvar = 35.247
```

Значення вершини стека ST(0) записується у операнд призначення dest. Після цього TOP збільшується на одиницю (причому, якщо TOP дорівнював 7, то буде TOP = 0) – так визначається нове розташування ST(0).

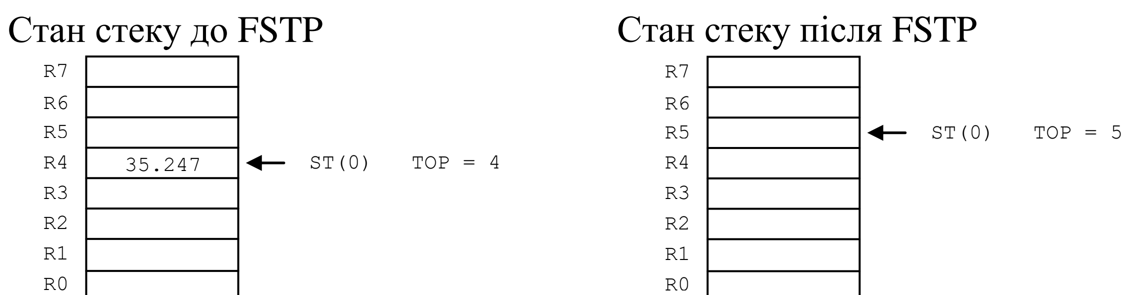


Рис. 4. Читання зі стеку одного числа командою FSTP

Операндом команди FSTP може бути 80-бітова, або 64-бітова, або 32-бітова перемінна, наприклад

```
Var32 dd ?
Var64 dq ?
Var80 dt ?
. . .
fstp dword ptr[Var32]
fstp qword ptr[Var64]
fstp tbyte ptr[Var80]
```

При читанні значення зі стеку FPU у 32-бітову перемінну, виконується перетворення у 32-бітовий формат Single-Precision з плаваючою точкою.

При читанні значення зі стеку FPU у 64-бітову перемінну, виконується перетворення у 64-бітовий формат Double-Precision з плаваючою точкою.

Якщо такі перетворення неможливі, то у регістр статусу записується відповідне бітове значення.

Прочитати дані зі стеку також можна й командою FST

```
FST dest
```

Ця команда відрізняється від FSTP тим, що не виштовхує значення зі стеку (pop), тобто не змінює вказівник вершини стеку ST(0). Ще одна відмінність: команда FST не може прочитати 80-бітове значення – можна тільки у 64- або 32-бітовому форматах.

Короткий огляд команд x87 FPU

Імена усіх команд x87 FPU починаються з букви ‘F’. Виділимо, у першу чергу ті команди, які будуть корисні для виконання завдань лабораторної роботи.

FADD/FADDP	Додавання
FSUB/FSUBP	Віднімання
FMUL/FMULP	Множення
FDIV/FDIVP	Ділення
FABS	Абсолютна величина
FSQRT	Квадратний корінь
FSIN	Синус
FCOS	Косинус
FSINCOS	Синус та косинус
FPTAN	Тангенс
FPATAN	Арктангенс
FYL2X	Логарифм ($y*\log_2 x$)
FYL2XP1	Логарифм ($y*\log_2 (x+1)$)
F2XM1	Експонента ($2^x – 1$)
FSCALE	Множення операнду на степінь 2

Докладні відомості про команди x87 FPU можна знайти у літературному джерелі від розробників процесорів: "Intel® 64 and IA-32 Architectures. Software Developer’s Manual".

Приклад обчислень

Розглянемо приклад обчислення у середовищі x87 FPU двох пар добутків $Res = A \times B + C \times D$. Процес обчислень проілюстровано на рис. 7.5.

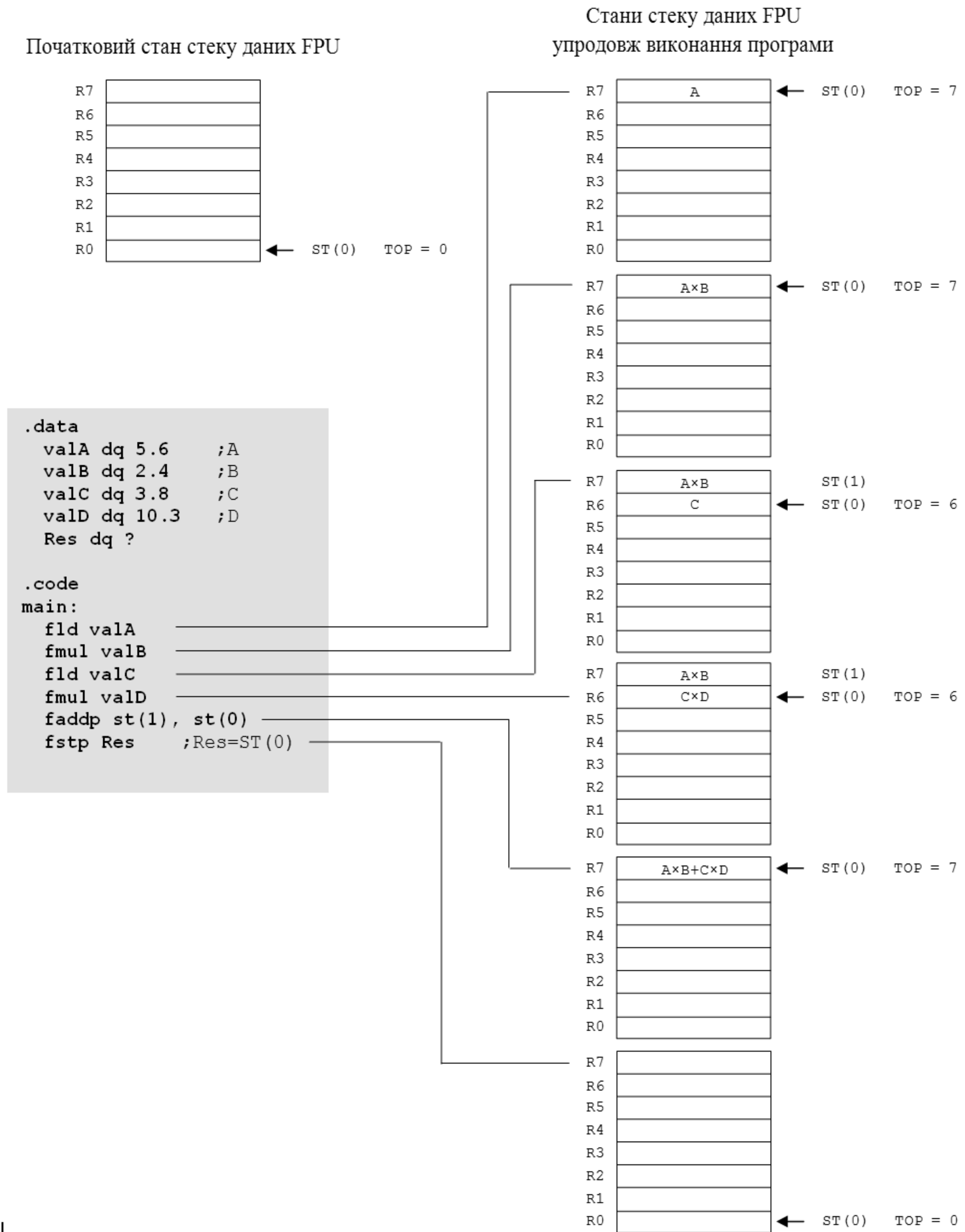


Рис. 7.5. Обчислення $Res = A \times B + C \times D$

Методичні рекомендації

Показ значень чисел, записаних у форматі з плаваючою точкою

Для того, щоб відображати результати обчислень, потрібно створити процедуру, яка перетворює значення, записане у форматі з плаваючою точкою, у текст дробу з десяткових цифр.

У лабораторній роботі №3 вже були розглянуті основні відомості щодо стандартних (IEEE754) двійкових форматів з плаваючою точкою. Тепер розглянемо деякі аспекти щодо переводу у десяткову систему числення.

Математично число з плаваючою точкою можна описати формулою

$$V = (-1)^S 2^E M,$$

де:

S – знак (0 означає плюс, 1 означає мінус);

E – експонента;

M – мантиса, яка дорівнює 1.F, де F – дробова частина мантиси

Приклад двійкового коду числа у 32-бітовому форматі з плаваючою точкою

$V = 420F12A1_{16} = 01000010000011110001001010100001_2$

Виділимо у цьому коді групи бітів елементів двійкового формату

S	e	F
0	10000100	00011110001001010100001
1	8	23

де:

S – біт знаку;

e – 8-бітовий зміщений код експоненти, причому e = E+127;

F – 23-бітова дробова частина мантиси.

Мантиса дорівнює **1.F = 1.00011110001001010100001**

Обчислимо експоненту E. Оскільки у форматі записується її зміщений код, тобто e = E + 127, тоді E = e – 127. Маємо $e = 10000100_2 = 132_{10}$, відповідно E = 132 – 127 = 5. Тоді число можна записати так:

$V = 2^5 \cdot 1.00011110001001010100001$

Запишемо це число без експоненти, як звичайне дробове число. Для цього помножимо мантису на 2^5, тобто, виконаємо зсув мантиси вліво на 5 бітів. Ліворуч точки буде ціла частина, праворуч – дробова

```
V = 100011.110001001010100001
```

Як перевести дробове двійкове число у десяткову систему? Таке завдання можна розчленувати на два кроки: окремо переводити у десятковий код цілу частину і окремо – дробову частину.

1. Перевести двійковий код цілої частини у десятковий можна послідовним **діленням на десять** у двійковому коді (двійковий код десяти є 1010_2):

$100011_2 : 1010_2 = 11_2$, залишок 101_2 – це десяткова цифра 5

$11_2 : 1010_2 = 0$, залишок 11_2 – це десяткова цифра 3

Таким чином, ціла частина у десятковому коді дорівнює 35.

2. Переведення у десятковий код дробової частини виконується **множенням на десять** у двійковому коді. Після кожного множення десяткова цифра є цілою частиною добутку. Можна порекомендувати швидкий алгоритм множення числа на 10 у двійковому коді: число (D) зсувається на біт вліво (2D) і до нього додається те саме число, зсунуте на три біти вліво (8D), тобто 10D = 2D+8D. Виконаємо це для конкретного двійкового коду дробової частини:

```
                       .110001001010100001    D1
                      1.10001001010100001     2D1
                    110.001001010100001       8D1
Перша цифра:  7 <-- [111].10101110100100101   10D1
              ----------------------------------
                       .10101110100100101     D2
                      1.0101110100100101      2D2
                    101.01110100100101        8D2
Друга цифра:  6 <-- [110].1101000110111001    10D2
              ----------------------------------
                       .1101000110111001      D3
                      1.101000110111001       2D3
                    110.1000110111001         8D3
Третя цифра:  8 <-- [1000].001100010011101    10D3
              ----------------------------------
```

```
                          .001100010011101        D4
                         0.01100010011101        2D4
                       001.100010011101          8D4
Четверта цифра:  1 <-- [001].11101100010001      10D4
-----------------------------------------------------
                          .11101100010001         D5
                         1.1101100010001         2D5
                       111.01100010001           8D5
П'ята цифра:  9 <-- [1001].0011101010101         10D5
-----------------------------------------------------
                          .0011101010101          D6
                         0.011101010101          2D6
                       001.1101010101            8D6
Шоста цифра:  2 <-- [010].010010101001           10D6
-----------------------------------------------------
```

Якщо обмежитися шістьма цифрами дробової частини, то отримаємо такий результат: V ≈ 35.768192

Особливі випадки у форматах з плаваючою точкою. Передбачено, що деякі комбінації бітів коду двійкового формату із плаваючою точкою використовуються для позначення особливих випадків – не число (*Not a Number – NaN*), безкінечність, нуль тощо. Ці особливі випадки позначаються кодами e = 00…0, а також e = 11…1 у таблиці 7.1.

Таблиця 7.1

Різновиди кодованих значень у форматах з плаваючою точкою

Зміщена експонента e = E + зсув	Дроб F	Значення числа V	Назва числа
$e = e_{min} - 1 = 00\ldots0$	F = 0	$V = (-1)^S\, 0$	нуль
	F ≠ 0	$V = (-1)^S\, 2^{Emin}\, (0.F)$	ненормалізоване число
$e_{min} \le e \le e_{max}$	F	$V = (-1)^S\, 2^E\, (1.F)$	нормалізоване число
$e = e_{max} + 1 = 11\ldots1$	F = 0	$V = (-1)^S\, \infty$	безкінечність зі знаком
	F ≠ 0	V = NaN	не число

Алгоритм дешифрування двійкового формату з плаваючою точкою

Для перетворення у десятковий запис двійкового коду 32-бітового формату з плаваючою точкою можна рекомендувати алгоритм дешифрування (аналізу) коду, наведений на рис. 7.6. Подібний алгоритм можна використати також для дешифрування 64-бітового та 80-бітового форматів.

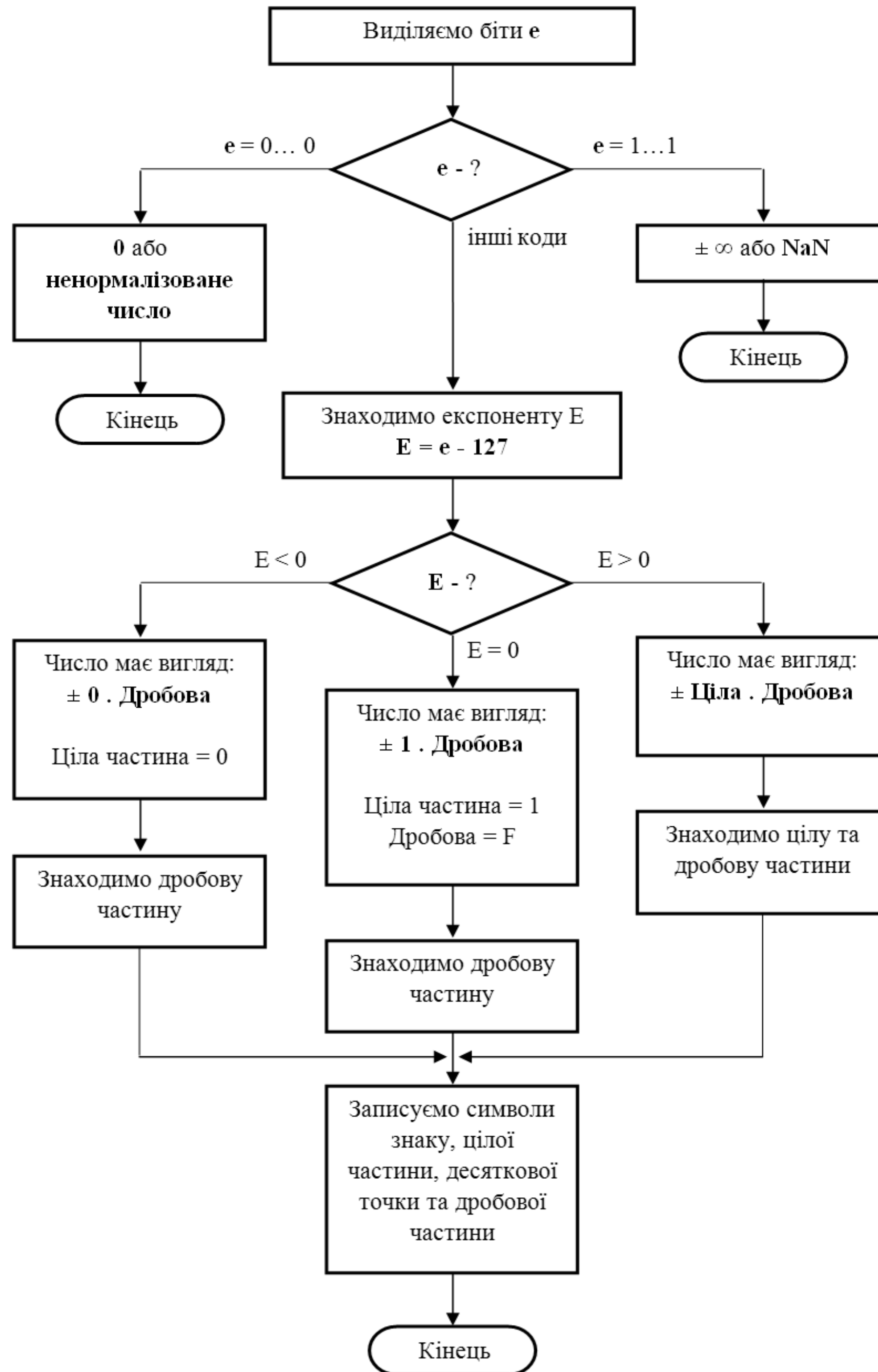


Рис. 7.6. Алгоритм дешифрування 32-бітового формату з плаваючою точкою

Варіанти завдань та основні вимоги

Запрограмувати обчислення математичного виразу на асемблері на основі команд x87 FPU. Номер варіанту відповідно номеру студента у журналі.

Таблиця 7.2

№ вар	Що потрібно обчислити	Формат даних для A,B,X
1	$Res = A_0 + A_1X + A_2X^2 + \ldots + A_{n-1} X^{n-1}$	64-бітовий
2	$Res = \mid A_0 B_0 + A_1 B_1 + \ldots + A_{n-1} B_{n-1} \mid$	32-бітовий
3	$Res = A_0 + A_1 \sin B + A_2 \sin(2B) + \ldots + A_n \sin(nB)$	64-бітовий
4	X_1, X_2 = Рішення системи лінійних рівнянь 2-го	32-бітовий
5	Res = квадратний корінь $(B^2{}_0 + B^2{}_1 + \ldots + B^2{}_{n-1})$	64-бітовий
6	$Res = A_0 + A_1 \sin X + A_2 \sin^2 X + \ldots + A_n \sin^n X$	32-бітовий
7	$Res = (A_0 B_0 + A_1 B_1 + \ldots + A_{n-1} B_{n-1}) / (A_0 + A_1 + \ldots +$	64-бітовий
8	$Res = A_1 (X–B_1) + A_2 (X–B_2)^2 + \ldots + A_n (X–B_n)^n$	32-бітовий
9	$Res = (\ldots(A_{n-1} X+A_{n-2}) X + A_{n-3}) X + \ldots A_1) X + A_0$	64-бітовий
10	$Res = B / (A_0X^2 + A_1X^2 + \ldots + A_{n-1} X^2)$	32-бітовий
11	$Res = A_0 + A_1 \cos B + A_2 \cos^2 B + \ldots + A_n \cos^n B$	64-бітовий
12	$Res = A_0 + A_1 \cos B + A_2 \cos(2B) + \ldots + A_n \cos(nB)$	32-бітовий
13	$Res = (X–A_0)^2 + (X–A_1)^2 + \ldots + (X–A_{n-1})^2$	64-бітовий
14	Res = логарифм $(A_0B_0 + A_1B_1 + \ldots + A_{n-1} B_{n-1})$	32-бітовий
15	$Res = \mid A_0 / B_0 + A_1 / B_1 + \ldots + A_{n-1} / B_{n-1} \mid$	64-бітовий
16	Res = Детермінант 3-го порядку	32-бітовий
17	$Res = A_0 + A_1 tgB + A_2 tg^2B + \ldots + A_n tg^nB$	64-бітовий
18	Res = максимальне значення $(A_i – B_i)^2$, для $i = 0, 1, . .$	32-бітовий
19	X_1, X_2 = Корені рівняння $AX^2 + BX + C = 0$	64-бітовий
20	Res = мінімальне значення $(A_i + B_i)^2$, для $i = 0, 1, . . .,$	32-бітовий
21	$Res = arctg(A_0 / B_0) + arctg(A_1 / B_1) + \ldots + arctg(A_{n-1} /$	64-бітовий
22	Res = мінімальне значення $(A_i – B_i)^2$, для $i = 0, 1, . . .,$	32-бітовий

Примітка. Вказані тести можуть бути викладачем замінені на інші тести спеціальним оголошенням перед видачею завдань студентам.

Результат надати у вікні MessageBox у вигляді десяткового дробу з цілої частини, точки та 7 розрядів дробової частини, отриманих з 32-бітового формату з плаваючою точкою. Перетворення з двійкового формату з

плаваючою точкою у рядок десяткових цифр оформити у вигляді процедури з ім'ям **FloatToDec**.

Заохочення: студентам, які запрограмують таку процедуру, яка виконує перетворення у текст десяткового дробу з 64-бітового або 80-бітового двійкових форматів з плаваючою точкою, може бути суттєво підвищена оцінка.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів та вихідного тексту
6. Висновки

Контрольні запитання

1. Як організований стек даних x87 FPU?
2. Що таке TOP?
3. Які команди забезпечують запис та читання даних у стек x87 FPU?
4. Що таке нормалізоване число?
5. Як кодується нуль, безкінечність, не-число у форматі з плаваючою точкою?
6. Як виділити з двійкового коду експоненту?
7. Як виділити з двійкового коду мантису?
8. Як виділити з двійкового коду цілу та дробові частини числа?
9. Як перевести дробове число з двійкового у десятковий код?

Лабораторна робота №8. Використання системних функцій у програмах на асемблері

Мета: Навчитися використовувати у програмах на асемблері функції операційної системи динамічного виділення пам'яті та запису файлів.

Завдання

1. Створити у середовищі Microsoft Visual Studio проект з ім'ям **Lab8**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути головний файл **main8.asm,** модуль **module** (за необхідності) та модуль **longop** попередніх робіт.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуваний файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – файл числових значень згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати та вихідний текст.

Теоретичні відомості

Написання на асемблері програм з викликами системних функцій було відоме ще давно, наприклад, у MS-DOS – там виклики системних функцій були оформлені як переривання INTnn.

Для забезпечення можливостей створювати програми, які працюватимуть у середовищі Windows, програмістам наданий інтерфейс API (Application Program Interface) у вигляді прототипів системних функції та інших бібліотечних файлів. При розробці власних програм програмісти можуть писати у вихідних текстах програмного коду виклики функцій API. У такий спосіб можна створювати програми на різноманітних мовах програмування, у тому числі на асемблері.

Виклик функцій API Windows у програмах на асемблері зручно програмувати за допомогою директиви INVOKE. Формат виклику

```
invoke Ім'яФункції, параметри
```

Програмування запису файлів

Для того, щоб записати щось у файл, спочатку потрібно відкрити цей файл, а якщо цього файлу ще немає – то створити його. Так, чи інакше, файл

відкривається і з ним зв'язується деякий ідентифікатор, так званий хендл (handle). Використовуючи цей хендл, можна викликати функції для запису у файл порцій інформації. Для відкриття або для створення файлу можна скористатися функцією CreateFile. Ця функція відкриває старий або створює новий файл відповідно до вказаних їй параметрів і записує хендл (*handle*) файлу у регістр EAX. Якщо значення хендлу не дорівнює INVALID_HANDLE_VALUE, то програмі дозволений доступ до файлу.

Після того, як отримано значення хендлу файлу, для запису інформації у цей файл можна використати функцію WriteFile. Після роботи з файлом необхідно його закрити, викликавши функцію CloseHandle. Нижче наведений приклад запису у файл рядка тексту

```
.data
  hFile dd 0
  pRes dd 0
  szFileName db "tmp.txt",0
  szTextBuf db "Рядок тексту, записаний у файл",0

.code
  . . .
  invoke CreateFile, ADDR szFileName,
                     GENERIC_WRITE,
                     FILE_SHARE_WRITE,
                     0, CREATE_ALWAYS,
                     FILE_ATTRIBUTE_NORMAL,
                     0
  cmp eax, INVALID_HANDLE_VALUE
  je @exit                         ;доступ до файлу неможливий
  mov hFile, eax
  invoke lstrlen, ADDR szTextBuf
  invoke WriteFile, hFile, ADDR szTextBuf, eax, ADDR pRes, 0
  invoke CloseHandle, hFile

@exit:
  . . .
```

Величини GENERIC_WRITE, FILE_SHARE_WRITE, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL є символічними константами, оголошеними у файлі windows.inc. У разі помилки відкриття файлу (наприклад, через вказування неправильного імені файлу) функція CreateFile повертає через регістр EAX значення INVALID_HANDLE_VALUE. Функція WriteFile має такі параметри: хендл файлу, адреса блоку даних, кількість байтів даних, які потрібно записати,

адреса перемінної, у яку буде записуватися інформація про коректність операції. Останній параметр у нашому випадку нульовий.

Для того, щоб записати у текстовий файл декілька рядків тексту, кожний рядок повинен завершуватися символами 13, 10. Ці коди можна дописувати окремим викликом функції WriteFile.

Вказування імені файлу

Для цього зручно скористатися стандартним діалоговим вікном Windows. Декілька часто уживаних діалогових вікон оформлені у бібліотеку діалогових вікон загального користування. Для доступу до функцій цієї бібліотеки потрібно підключити файли **comdlg32.inc, lib**.

Для появи стандартного діалогового вікна відкриття файлу для запису треба викликати функцію **GetSaveFileName**. Перед викликом цієї функції необхідно створити структуру типу OPENFILENAME, заповнити поля цієї структури потрібними значеннями, а потім викликати функцію GetSaveFileName, передавши у якості параметру адресу структури. Необхідно відзначити, що подібний спосіб виклику функцій використовується для багатьох функції API Windows – замість передачі великої кількості параметрів через стек створюється блок даних, структура, адреса якої потім вказується при виклику.

Оформимо вибір імені файлу у вигляді окремої процедури MySaveFileName. Оскільки структура типу OPENFILENAME містить тимчасову інформацію, то можна її запрограмувати, наприклад, як локальну структуру.

```
include \masm32\include\comdlg32.inc
includelib \masm32\lib\comdlg32.lib
.data
  szFileName db 256 dup(0)                         ; буфер для імені файлу
.code
MySaveFileName proc
  LOCAL ofn : OPENFILENAME
  invoke RtlZeroMemory, ADDR ofn, SIZEOF ofn       ; спочатку усі поля
обнулюємо
  mov ofn.lStructSize, SIZEOF ofn
  mov ofn.lpstrFile, OFFSET szFileName
  mov ofn.nMaxFile, SIZEOF szFileName
  invoke GetSaveFileName,ADDR ofn                  ; виклик вікна File Save As
  ret
MySaveFileName endp
```

```
main:
  . . .
  call MySaveFileName
  cmp eax, 0              ; перевірка: якщо у вікні було натиснуто кнопку Cancel, то EAX=0
  je @exit
```

Для ініціалізації деяких полів використана директива SIZEOF, яка обчислює на етапі компіляції розмір деякого об’єкту у байтах.

Використання глобальної динамічної пам'яті

Для організації масивів даних особливо тих, які впродовж роботи програми створюються тимчасово (динамічно), зручно використовувати глобальну пам’ять, яку надає ОС по запиту від програм. Якщо казати точніше, то виділяється пам'ять у віртуальній памяті, яку створює та керує операційна система, і частку з неї дозволяє використовувати програмам. Обсяг такої пам'яті для кожної прикладної програми Win32 обмежується двома гігабайтами, для Win64 – обсяг значно більший.

Для створення динамічного масиву з глобальної пам'яті можна використати функцію **GlobalAlloc**, яка належіть до складу API Windows. Ця функція у залежності від параметрів виклику може повертати вказівник – адресу блоку пам'яті, що виділяється. Потім цей вказівник використовується для запису або читання потрібних даних. Коли динамічний масив стає непотрібним, його треба знищити за допомогою функції **GlobalFree**.

```
.data
  pD dd ?                          ; оголошуємо вказівник як 32-бітову перемінну

.code
  . . .
  invoke GlobalAlloc, GPTR, 1024   ; створюємо динамічний масив з 1024 байт
  mov pD, eax                      ; беремо вказівник з EAX – результат GlobalAlloc
  . . .                            ; працюємо з динамічним масивом
  invoke GlobalFree, pD            ; знищуємо динамічний масив - звільняємо пам'ять
```

Вказування параметру GPTR для GlobalAlloc означає, що ця функція повертає адресу виділеного блоку пам’яті, а також цей блок від початку заповнюється нулями.

Методичні рекомендації

Запис масиву значень факторіалу. Перший варіант циклу

Відповідно завдання у даній лабораторній роботі використовуються активи, зроблені в результаті виконання ЛР №4 щодо обчислення факторіалу. Нехай маємо процедуру обчислення факторіалу, яка зветься, наприклад, **Factorial**. Для запису масиву значень $n!$ (n від 1 до n_{max}) потрібно у циклі n_{max} разів викликати цю процедуру. На кожному кроці такого циклу результат – двійковий код підвищеної розрядності перетворюється у рядок десяткових цифр і записується у файл. Цей алгоритм можна відобразити так:

```
n = 1
while (n <= nmax)
  {
  Factorial(n, fact)       //процедура обчислює n! Результат записує у fact
  StrDec(fact, textbuf)    //перетворення двійкового коду у десятковий
  WriteFile(textbuf)       //запис рядка десяткових цифр
  n = n+1
  }
```

Оформлення обчислення факторіалу у вигляді окремої самодостатньої процедури сприяє структурованості програмного коду, проте має суттєвий недолік – багаторазове виконання зайвих операцій множення підвищеної розрядності, оскільки кожний виклик процедури Factorial означає послідовне множення чисел від 1 до n.

Другий варіант циклу

Він побудований вже як цикл послідовних множень, і після кожного множення його результат одразу записується у файл у якості значення факторіалу.

```
fact = 1
n = 1
while (n <= nmax)
  {
  fact = fact * n     //множення числа підвищеної розрядності (fact) на 32-бітове (n)
  StrDec(fact,textbuf)    //перетворення двійкового коду у десятковий рядок
  WriteFile(textbuf)      //запис десяткового рядка цифр
  n = n+1
}
```

У такому варіанті циклу операцій множення виконується загалом стільки, скільки необхідно.

Статичні та динамічні дані у лабораторній роботі

Розглянемо приклад обчислення факторіалу зі статичними даними

```
.data
  Result dd 32 dup(0)        ; статичний масив Result 1024 байтів
  val dd 32 dup(0)           ; статичний масив val
  n dd 1
.code
main:
  mov dword ptr[val], 1      ; val = 1
@cycle:
  inc dword ptr [n]          ; n = n+1
  mov eax, dword ptr [n]
  cmp eax, 50                ; 50!
  jg @endfactorial

  push offset val
  push eax                   ; n
  push offset Result
  push 256
  call MulTo32_LONGOP        ; Result = val * n

  push offset Result
  push offset val
  push 256
  call Copy_LONGOP           ; val <- Result
  jmp @cycle
```

А тепер замість статичного 1024-байтового масиву val створимо тимчасовий динамічний масив, адресу якого буде зберігати вказівник pVal. Після обчислення факторіалу цей масив, який зберігає тимчасові значення підвищеної розрядності, стає непотрібним і він знищується.

```
.data
  Result dd 32 dup(0)        ; статичний масив
  pVal dd ?                  ; це буде вказівник на тимчасовий динамічний масив val
  n dd 1

.code
main:
  invoke GlobalAlloc, GPTR, 1024
  mov pVal, eax
  mov dword ptr[eax], 1      ; val = 1 (запис у масив по вказівнику робиться через регістр)
@cycle:
```

```
  inc dword ptr [n]             ; n = n+1
  mov eax, dword ptr [n]
  cmp eax, 50                   ; 50!
  jg @endfactorial

  push pVal
  push eax                      ; n
  push offset Result
  push 256
  call MulTo32_LONGOP           ; Result = val * n

  push offset Result
  push pVal
  push 256
  call Copy_LONGOP              ; val <– Result

  jmp @cycle

@endfactorial:
  invoke GlobalFree, pVal       ; знищуємо динамічний масив - звільняємо пам'ять
  . . .
```

Червоним виділено рядки коду, які стосуються тимчасового динамічного масиву val.

Зверніть увагу на те, що вказівник pVal містить адресу потрібного об’єкту, тому процедурам MulTo32_LONGOP та Copy_LONGOP у відповідних параметрах передаються не "offset ім'я обєкту", а значення вказівника.

Звісно, так само замість статичного масиву Result можна запрограмувати роботу ще з одним відповідним динамічним масивом. Здавалося б, замість двох статичних масивів val та Result можна оголосити два вказівника, наприклад, pVal та pRes та двічі викликати функцію GlobalAlloc, і відповідно, для знищення двох масивів потрібно буде двічі викликати функцію GlobalFree. А можливо зробити інакше – замість багатьох маленьких масивів виділити динамічний блок пам’яті, у якому будуть розташовані декілька структур даних. Замість двох масивів по 1024 байтів створимо один блок пам’яті 2048 байтів, у якому виділимо адреси потрібних масивів. Такі адреси можна зберігати і у окремих перемінних-вказівниках.

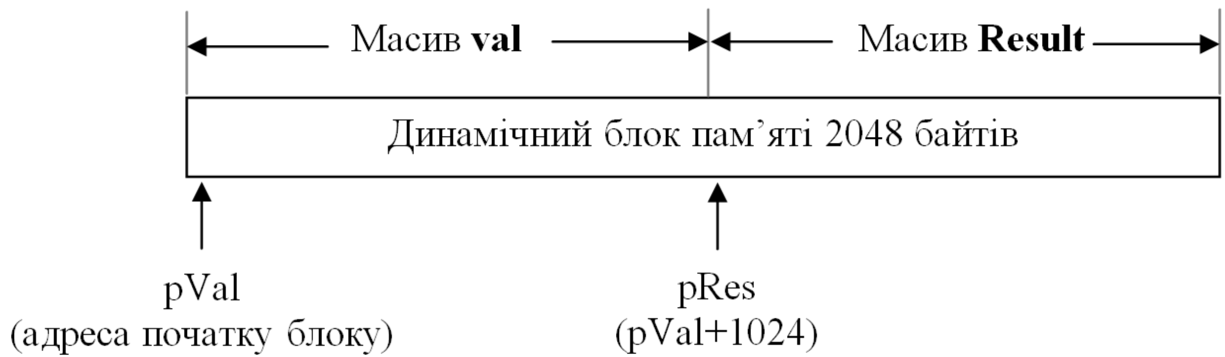


Рис. 8.1. Блок пам'яті для двох масивів

Приклад реалізації

```
.data
  pRes dd ?                  ; це буде вказівник на тимчасовий динамічний масив Result
  pVal dd ?                  ; це буде вказівник на тимчасовий динамічний масив val
  n dd 1

.code
main:
  invoke GlobalAlloc, GPTR, 2048
  mov pVal, eax
  mov dword ptr[eax], 1      ; val = 1 (запис у масив по вказівнику робиться через регістр)
  add eax, 1024              ; адреса масиву Result
  mov pRes, eax
@cycle:
  inc dword ptr [n]          ; n = n+1
  mov eax, dword ptr [n]
  cmp eax, 50                ; 50!
  jg @endfactorial

  push pVal
  push eax                   ; n
  push pRes
  push 256
  call MulTo32_LONGOP        ; Result = val * n

  push pRes
  push pVal
  push 256
  call Copy_LONGOP           ; val <– Result

  jmp @cycle

@endfactorial:
  invoke GlobalFree, pVal    ; знищуємо динамічний блок пам'яті
  . . .
```

Зауваження. Чи обов'язково для обчислення факторіалу треба відкривати два масиви? Якщо зробити процедуру **MulTo32_LONGOP** такою, що дозволяє результат множення записувати на місце операнду підвищеної розрядності, тоді не потрібно копіювати результат і програма може суттєво спроститися. Спробуйте це.

Варіанти завдань та основні вимоги

Запрограмувати на асемблері запис у файл масиву значень факторіалу $n!$ (n від 1 до n_{max}) відповідно табл. 8.1. Вказування імені файлу у стандартному діалоговому вікні (функція GetSaveFileName). Для кожного студента своє значення n_{max}

$$n_{max} = 30 + 2 \times H,$$

де H – це номер студента у журналі.

Таблиця 8.1

№ вар	Варіант циклу факторіалу	Код результату	Масиви для даних підвищеної розрядності	Масив-буфер для імені файлу
1	1	десятковий	динамічні	статичний
2	2	десятковий	динамічні	статичний
3	1	шістнадцятковий	динамічні	статичний
4	2	шістнадцятковий	динамічні	статичний
5	1	десятковий	статичні	динамічний
6	2	десятковий	статичні	динамічний
7	1	шістнадцятковий	статичні	динамічний
8	2	шістнадцятковий	статичні	динамічний
9	1	десятковий	динамічні	статичний
10	2	десятковий	динамічні	статичний
11	1	шістнадцятковий	динамічні	статичний
12	2	шістнадцятковий	динамічні	статичний
13	1	десятковий	статичні	динамічний
14	2	десятковий	динамічний	статичний
15	1	шістнадцятковий	статичні	динамічний
16	2	шістнадцятковий	динамічний	статичний
17	1	десятковий	динамічні	динамічний

18	2	десятковий	динамічні	динамічний
19	1	шістнадцятковий	динамічні	статичний
20	2	шістнадцятковий	динамічні	статичний
21	1	десятковий	статичні	динамічний
22	2	десятковий	статичні	динамічний
23	1	шістнадцятковий	статичні	динамічний
24	2	шістнадцятковий	статичні	динамічний
25	1	десятковий	динамічні	статичний
26	2	десятковий	динамічні	статичний
27	1	шістнадцятковий	динамічні	статичний
28	2	шістнадцятковий	динамічні	статичний
29	1	десятковий	динамічні	статичний
30	2	десятковий	динамічні	статичний

Примітка. Параметри варіантів завдань можуть бути змінені викладачем шляхом оголошення студентам відповідного повідомлення завчасно перед постановкою завдань.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів та вихідного тексту
6. Висновки

Контрольні запитання

1. Що таке API Windows?
2. Як викликати системну функцію ОС Windows?
3. Як забезпечити використання системної функції у програмному коді?
4. Що таке хендл?
5. Як відкрити файл для запису даних?
6. Яка функція записує у файл?
7. Як створити динамічні масиви даних?

Лабораторна робота №9. Використання у проекті C++ модулів на асемблері

Мета: Навчитися програмувати модулі на асемблері, у яких містяться команди команди SSE, x87 FPU а також використовувати такі модулі у проектах C++.

Завдання

1. Створити проект Windows Desktop Application з ім'ям Lab9.

2. Написати на асемблері процедуру обчислення скалярного добутку двох векторів із використанням команд SSE. Ім'я процедури: **MyDotProduct_SSE**. Процедуру оформити у окремому модулі і записати файли vectsse.asm, vectsse.h. Додати файл vectsse.asm у проект.

3. Запрограмувати на асемблері процедуру обчислення скалярного добутку двох векторів на основі команд x87 FPU без використання команд SSE. Ім'я процедури: **MyDotProduct_FPU**. Процедуру оформити у окремому модулі і записати файли vectfpu.asm, vectfpu.h. Додати файл vectfpu.asm у проект.

4. Запрограмувати на C++ обчислення скалярного добутку тих самих векторів як звичайну функцію C++ з ім'ям **MyDotProduct**, яка приймає значення двох масивів і записує результат у числову перемінну (будь-яка оптимізація при компіляції повинна бути відсутня).

5. Зробити меню для вікна програми так, щоб користувач програми мав можливість викликати процедури на асемблері MyDotProduct_SSE, MyDotProduct_FPU з модулів vectsse, vectfpu, а також функцію MyDotProduct.

6. Запрограмувати вивід результатів обчислень та виміри часу виконання скалярного добутку для трьох варіантів реалізації.

7. Отримати дизасемблерний текст функції C++ MyDotProduct. Проаналізувати код дизасемблеру, порівняти з кодом на асемблері процедури MyDotProduct_FPU.

8. Зробити висновки щодо використання модулів на асемблері у проектах програм, створених на основі мови C++ .

Теоретичні положення

Проекти з модулями на асемблері

У проектах програм на мовах високого рівня, зокрема на мові C++, є можливості використання модулів, написаних на асемблері. У таких модулях можуть міститися процедури, які будуть викликатися з інших модулів, написаних мовою C++. Наприклад, у проекті Lab9 у головному файлі Lab9.cpp запрограмовані виклики процедур, які містяться у файлах *.asm. У цьому випадку потрібно знати імена та типи цих процедур, типи їхніх аргументів. Така інформація записується у файли заголовків *.h.

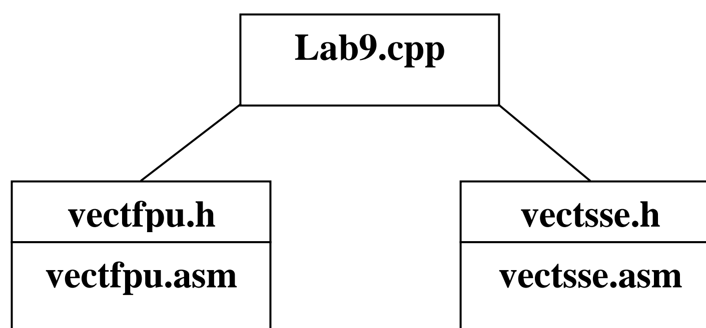


Рис. 9.1. Взаємодія модулів у проекті на C++

Конвенції виклику

Конвенція виклику – це формат побудови машинного коду, який генерує компілятор мови високого рівня для виклику процедур (функцій, методів). Конкретна конвенція виклику описує, як процедура приймає параметри, як видає результат і які дії потрібно виконати після відпрацювання процедури.

Конвенція stdcall прийнята для функцій API Windows. Параметри передаються через стек, у порядку, зворотному їхньому запису, справа наліво (RTL – *Right To Left*). Стек відновлює (значення вказівника стеку) сама процедура.

Конвенція **cdecl** (від "*c-declaration*"). Цей спосіб прийнятий для мов C, C++. Параметри передаються через стек, у порядку RTL. Для 32-бітових програм аргументи, розмір яких менше 4-х байт, розширюються до 4-х байт. Відновлення стеку виконує той, хто викликає. Після відпрацювання процедури той, хто викликав цю процедуру, відновлює стек. Це зручно для створення процедур з перемінною кількістю параметрів а також при

оптимізації програмного коду компілятором. Якщо процедура повертає результат, то вона записує його у регістр EAX (якщо результат цілого типу), або у регістрі ST(0) стеку FPU (якщо тип результата з плаваючою точкою).

Таблиця 9.1

Характеристики конвенції cdecl

Тип	**Розмір значення у байтах**	**Спосіб передачі результату**	**Примітка**
Ціле число, вказівник	1, 2, 4	Через регістр EAX	Значення, розмір яких менше 32 бітів, розширюються до 32 бітів
Ціле число	8	Через пару регістрів EDX : EAX	
Число з плаваючою точкою	4, 8	Через регістр ST(0) стека x87 FPU	
Інші	Більше 8	Через регістр EAX	У регістрі EAX записується вказівник на блок даних

Програмісту, який створює на асемблері процедури, які планується використовувати в проектах C/C++, необхідно дотримуватися конвенції виклику, прийнятій у даному середовищі розробки. У нашому випадку cdecl.

Скалярний добуток векторів

Для n-вимірних векторів $\boldsymbol{A} = \{a_0, a_1, \ldots, a_{n-1}\}$ та $\boldsymbol{B} = \{b_0, b_1, \ldots, b_{n-1}\}$ скалярним добутком є число, яке дорівнює сумі попарних добутків елементів: $x = a_0\, b_0 + a_1\, b_1 + \ldots + a_{n-1}\, b_{n-1}$

Обчислення скалярного добутку векторів є базовою операцією для багатьох галузей – матричних обчислень, фільтрації, графіки, кореляційного аналізу тощо.

Якщо елементами векторів можуть бути дробові числа, то обчислення скалярного добутку можна запрограмувати на основі команд з плаваючою точкою блоку x87 FPU. А якщо поставити за мету максимальне прискорення обчислень, то можна порекомендувати для обробки векторів використовувати SIMD команди SSE, AVX.

Технологія SIMD

Абревіатура SIMD (*Single Instruction Multiple Data*) позначає наявність та апаратну підтримку команд спеціального типу – одна така команда обробляє одночасно множину значень (вектор).

Уявимо собі, що потрібно обробити в циклі, скажімо, 1024 значень масивів X[ ], Y[ ] і записати результати у масив R[ ]

```
float X[1024];        //вектор-джерело
float Y[1024];        //вектор-джерело
float R[1024];        //вектор-результат
for (int i=0; i<1024; i++)
    R[i] = X[i] op Y[i];
```

У цьому псевдокоді позначено деяку двомісну операцію як **op**. Це може бути множення, додавання тощо. Якщо замість такого циклу представити це у вигляді одної операції одразу над 1024 парами чисел

```
R = X op Y
```

то можна назвати таку операцію **векторною**. Векторна операція виконується над векторами і результатом є вектор. На відміну цього, деякі операції виконуються теж над векторами, але результатом є одне число. Такі операції звуться **скалярними** (наприклад, скалярний добуток векторів).

Якщо в циклі послідовно виконуються звичайні двомісні команди, то час обробки, наприклад 1024 пар можна приблизно оцінити як 1024*(час роботи одної команди). Примітка – без урахування можливостей певного процесора щодо розпаралелювання обчислювальних процесів.

Уявимо собі, що існує така векторна команда **op8**, яка сама обробляє одночасно не 2 а 8 пар операндів-векторів, записуючі 8 чисел у вектор результату – і це за один такт. Тоді можна переписати цикл наступним чином

```
for (int i=0; i<1024; i+=8)
    R[i,.., i+7] = X[i,.., i+7] op8 Y[i,.., i+7];
```

Можна сподіватися, що такий цикл буде виконаний у 8 разів швидше. Головна ідея SIMD: чим більше елементів векторів обробляється одною командою (за одни такт), тим більшим очікується прискорення порівняно з використанням двомісних команд.

Методичні рекомендації

Створення проекту С++ у середовищі MS Visual Studio

Для лаб9 це розпочинається так само, як і для попередніх ЛР. Для створення нового проекту виберіть в стартовій панелі “Create a new project”.

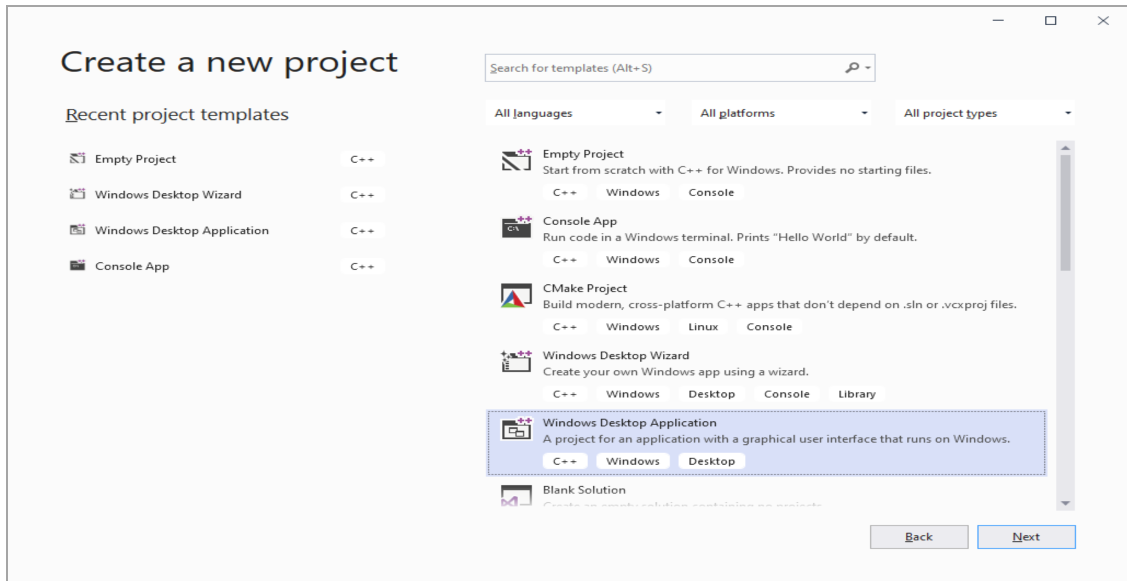


Рис. 9.1. Вибір типу проєкта, який потрібно створити

Наступні кроки вже дещо відрізняються від попередніх ЛР. Потрібно вибрати тип проекту “Windows Desktop Application”. У наступному вікні вказуємо ім’я проекту та розташування на диску

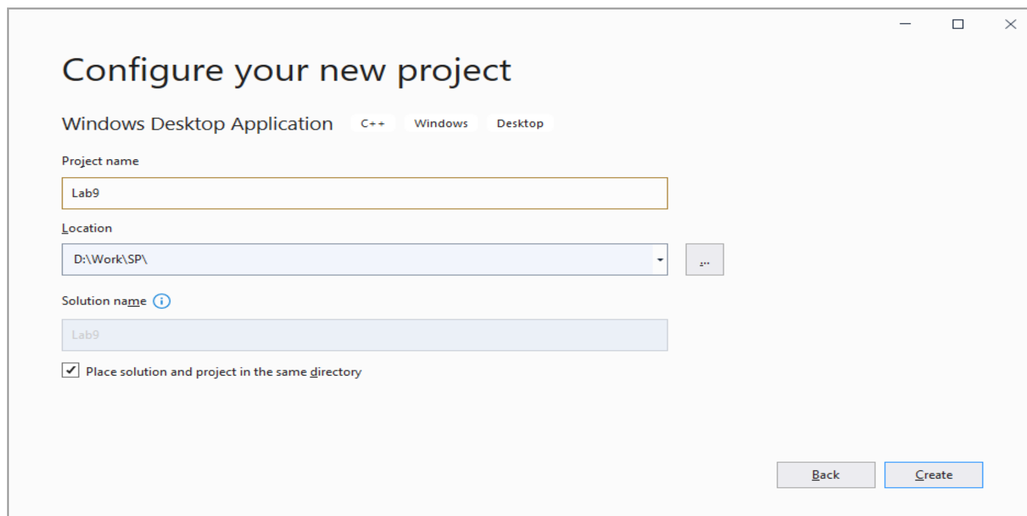


Рис. 9.2. Вказування імені проекту і його розташування на диску

Натиснемо кнопку “Create”. Visual Studio розпочне готувати файли проекту. Потім покаже новостворений проект.

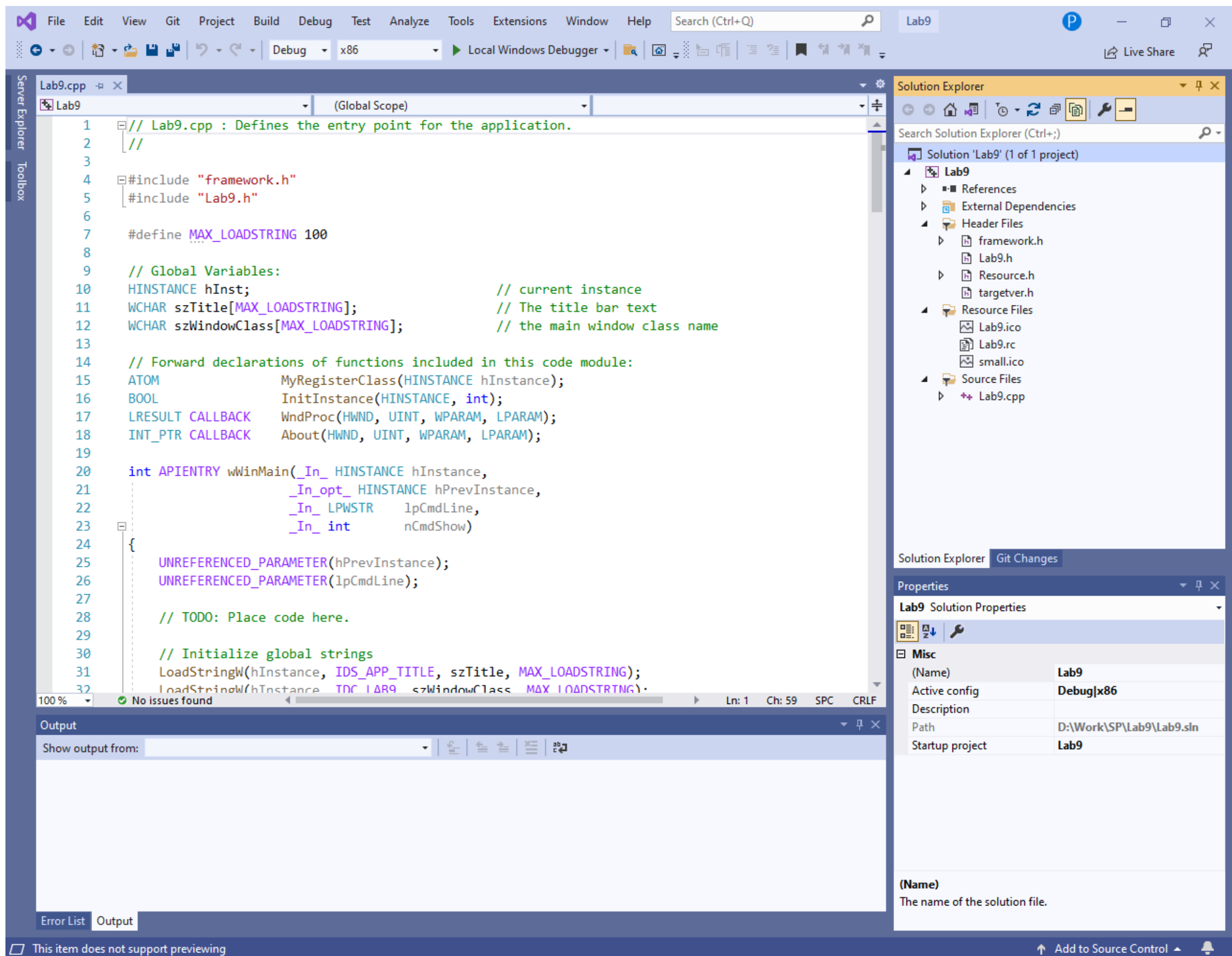


Рис. 9.3. Автоматично створений шаблон проекту Desktop Application

Visual Studio сгенерував початковий набір файлів для проекту С++. Імена цих файлів записані у розділах Source Files, Resource Files та Header Files. Список цих файлів можна побачити у вікні Solution Explorer праворуч.

Як і для будь-якого проекту С++, створюється головний файл проекту, який містить функцію **main** – точку входу у програму. Для проектів “Windows Desktop Application” головна функція зветься вже як WinMain

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                     _In_opt_ HINSTANCE hPrevInstance,
                     _In_ LPWSTR    lpCmdLine,
                     _In_ int       nCmdShow)
{
. . .
}
```

У шаблоні “Windows Desktop Application” передбачено програмування графічного інтерфейсу користувача. І деякі основні елементи цього інтерфейсу вже від початку Visual Studio створює автоматично. Що це за інтерфейс користувача і які його елементи?

1. Головне вікно програми
2. Меню у головному вікні
3. Вікно діалогу “About”

Варто подивитися, яку програму було автоматично створено від початку. Для цього виберемо меню “Start Debugging” – тоді Visual Studio скомпілює проект та викличе програму на виконання

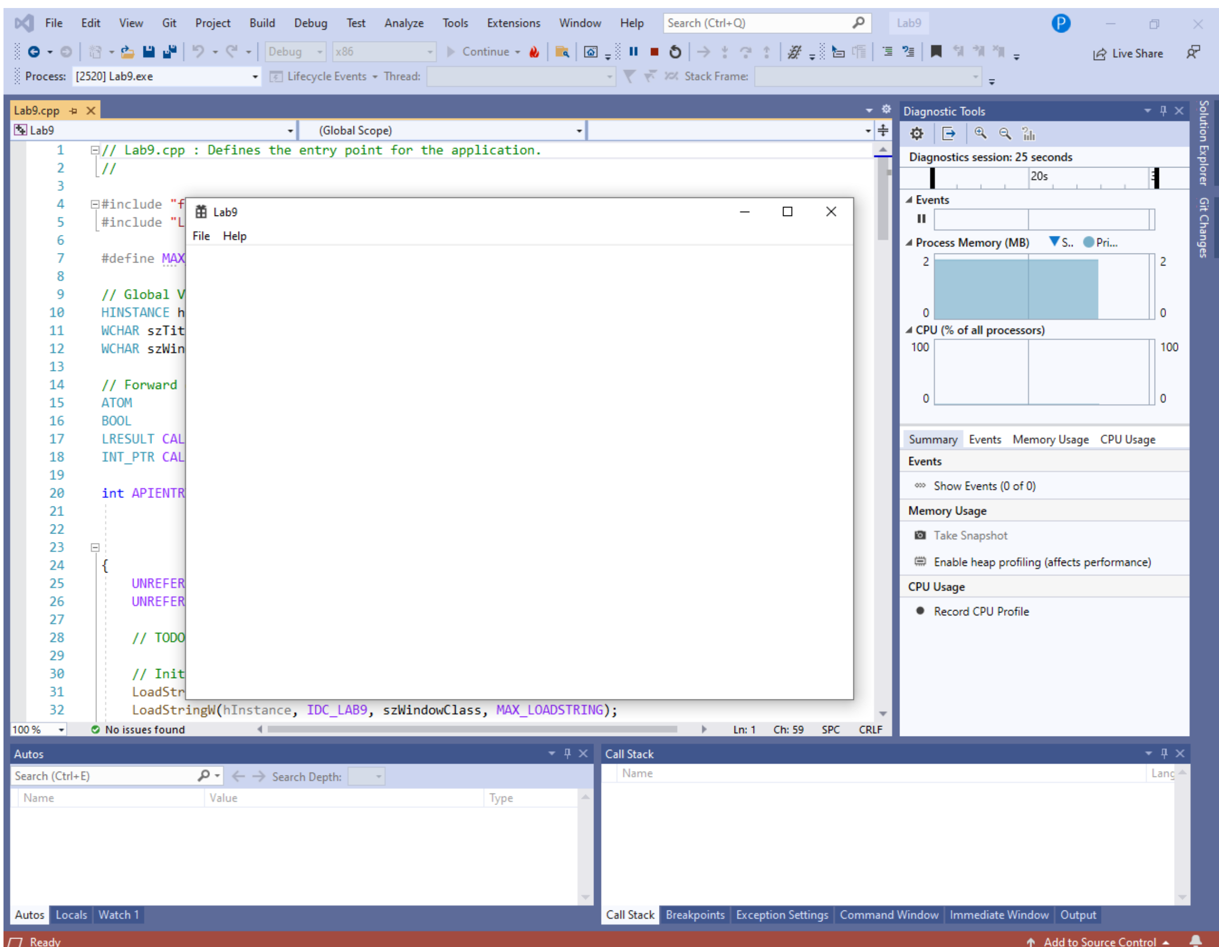


Рис. 9.4. Вигляд вікна новоствореної програми

Програма демонструє порожнє вікно з двома пунктами меню – File та Help.

Закриємо програму. Таким чином, будемо вважати, що Visual Studio сгенерував відповідно шаблону деякий початковий код, який далі програмісту треба дописувати, додавати відповідно конкретних вимог.

Налаштування підтримки мови асемблеру у проекті

Таке налаштування виконується так само, як і у попередніх ЛР – у вікні "Solution Explorer" клацніть правою кнопкою миші імені проекту. У спливаючому меню виберіть пункт "Build Customization".

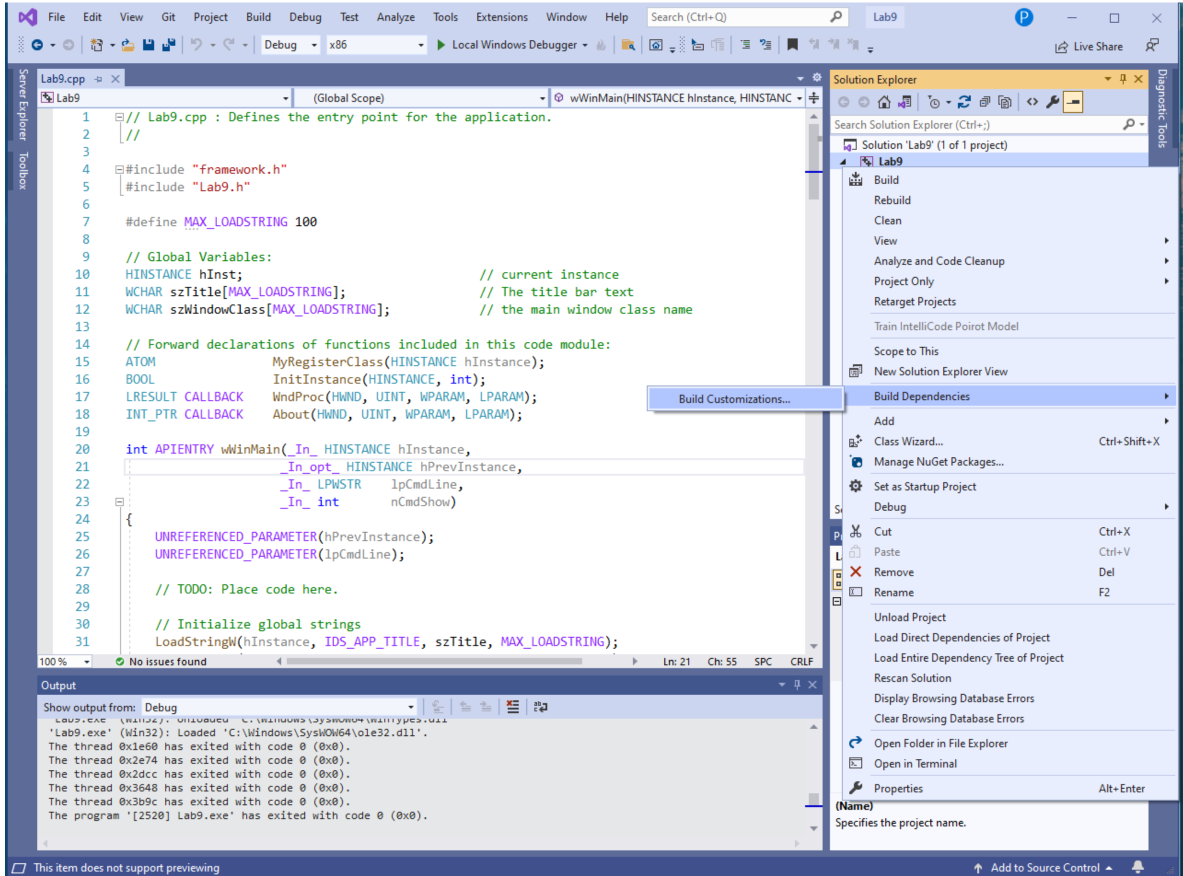


Рис. 9.5. Вибір Build Customization

А потім у діалоговому вікні треба вибрати опцію **masm**

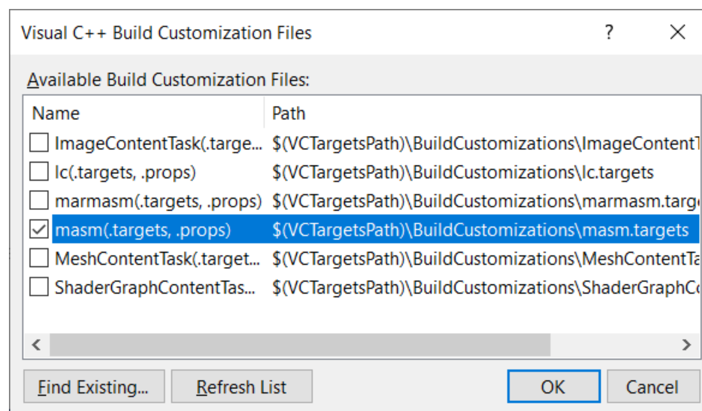


Рис. 9.6. Підключення асемблера у проєкт

Додавання модулів у проект

Потрібно у проект додати файли **vectsse.asm** та **vectfpu.asm** у розділ Source Files. Це виконується так само, як у попередніх лабах – правою кнопкою миші на назві розділу Source Files і далі у спливаючому меню вибрати пункт New Item…

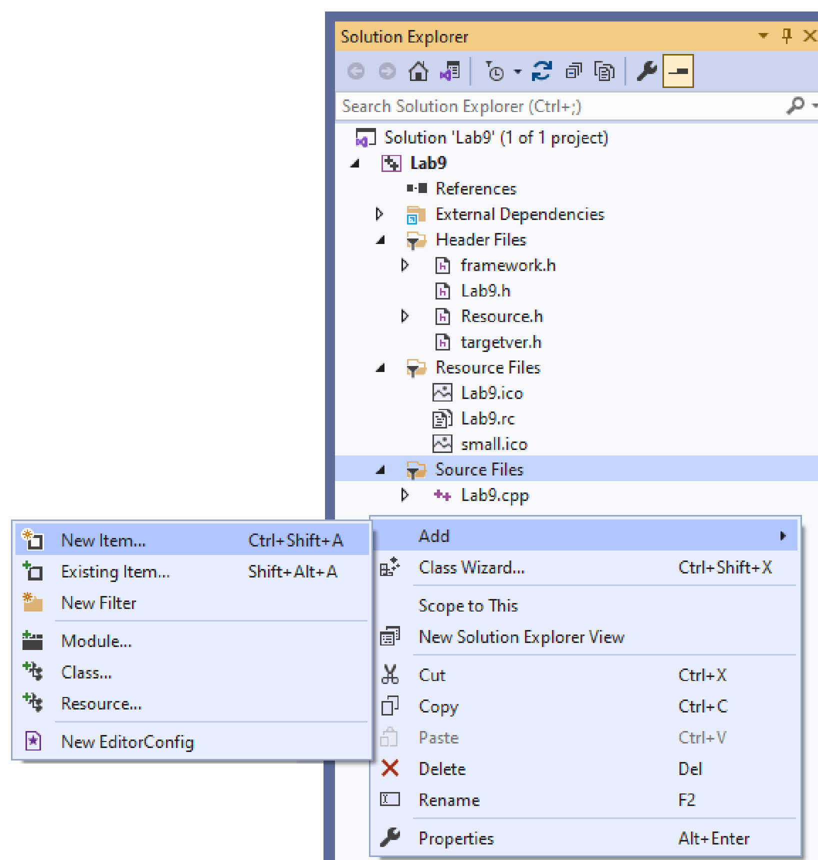


Рис. 9.7.Додавання файлів модулів у проєкт

Після цього у діалоговому вікні треба вписати **vectsse.asm.** Так само потім вставити у проект новий файл **vectfpu.asm.**

Файли заголовків – **vectsse.h** та **vectfpu.**h рекомендується також включити у проект – вже у розділ Header Files.

Сформувавши у такий спосіб модульну архітектуру проекту, треба переходити до наступних кроків його реалізації. Який буде наступний крок? Можна було б розпочати писати вихідний текст модулів на асемблері. Але у процесі їхньої розробки, вочевидь, треба буде їх тестувати та налагоджувати. Тоді постає питання: як можливо викликати з модулів якісь процедури задля їхнього тестування та налагодження? Для цього ще не готовий інтерфейс

користувача. Можна порекомендувати іншій підхід – проектування "з гори до низу". Що це означає?

Відповідно вимог має бути щоб програма Lab9 була керована через меню, тобто потрібні функції активізувалися би при виборі користувачем відповідних пунктів меню. Тоді можна спочатку створити і налагодити відповідний набір елементів керування – пунктів меню, а реакцію на їхній вибір запрограмувати пізніше. Підхід до проектування "з гори до низу" тут можна тлумачити як рух від опису загального функціоналу с точки зору вимог користувача до його усіх найменших подробиць конкретної реалізації і втілення.

Створення меню

Опис меню знаходиться у файлі ресурсів – Lab9.rc. Щоб внести потрібні зміни у меню можна скористатися вбудованим у Visual Studio редактором ресурсів. Щоб його активувати, треба двічі клікнути на файлі Lab9.rc у вікні Solution Explorer.

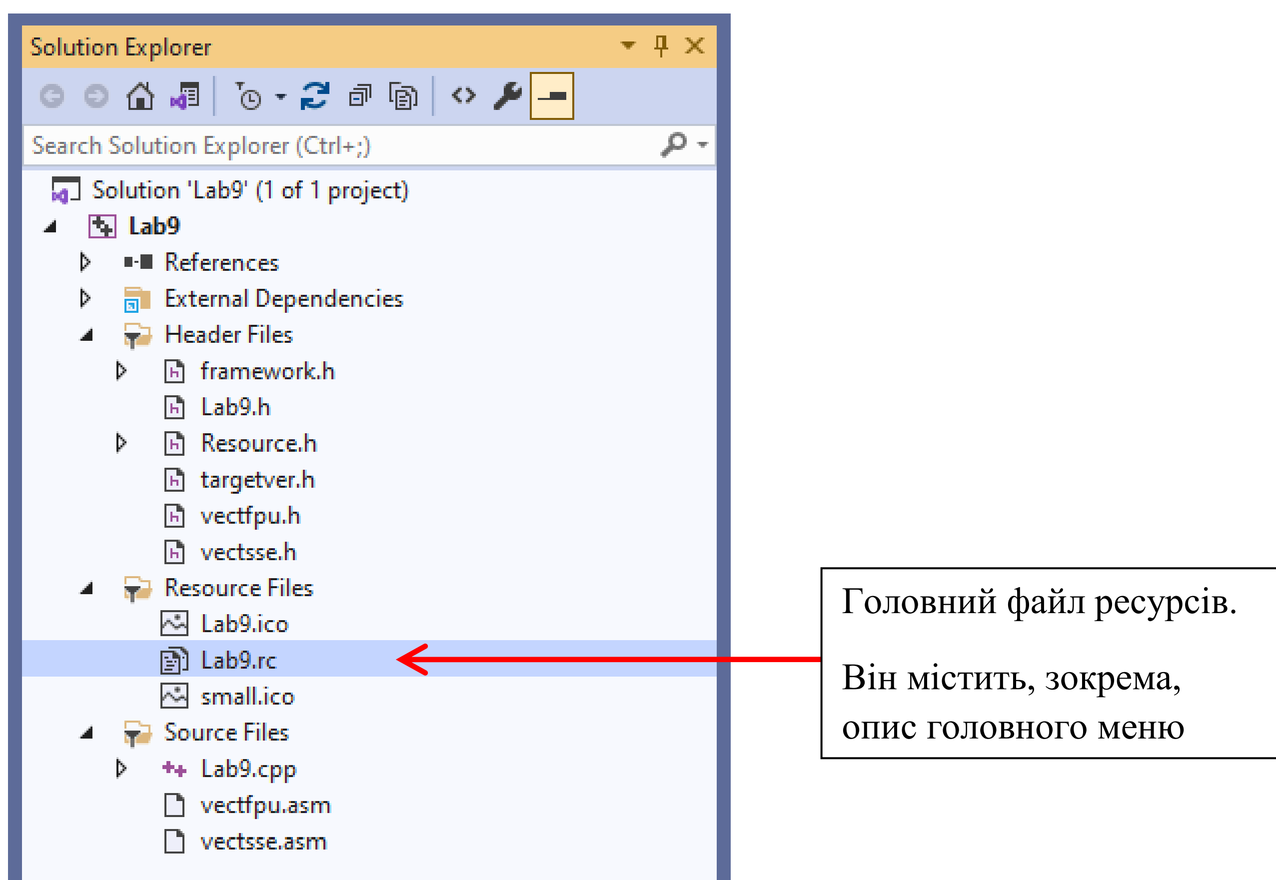


Рис. 9.8. Головний файл ресурсів у списку файлів проєкту

А далі у вікні Resource View треба вибрати рядок меню

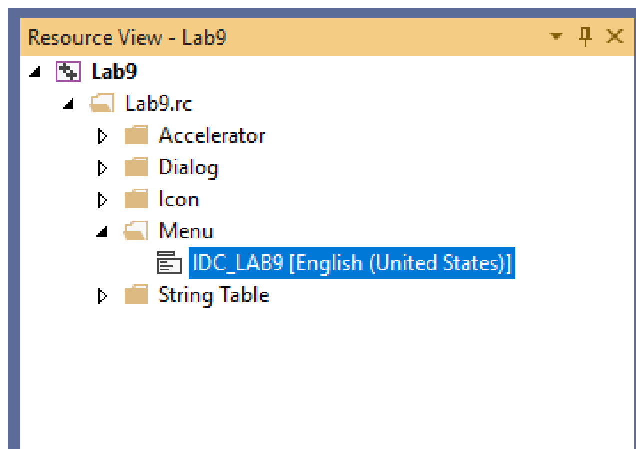


Рис. 9.9. Ресурс головного меню

Потім з'являється вікно редактора меню

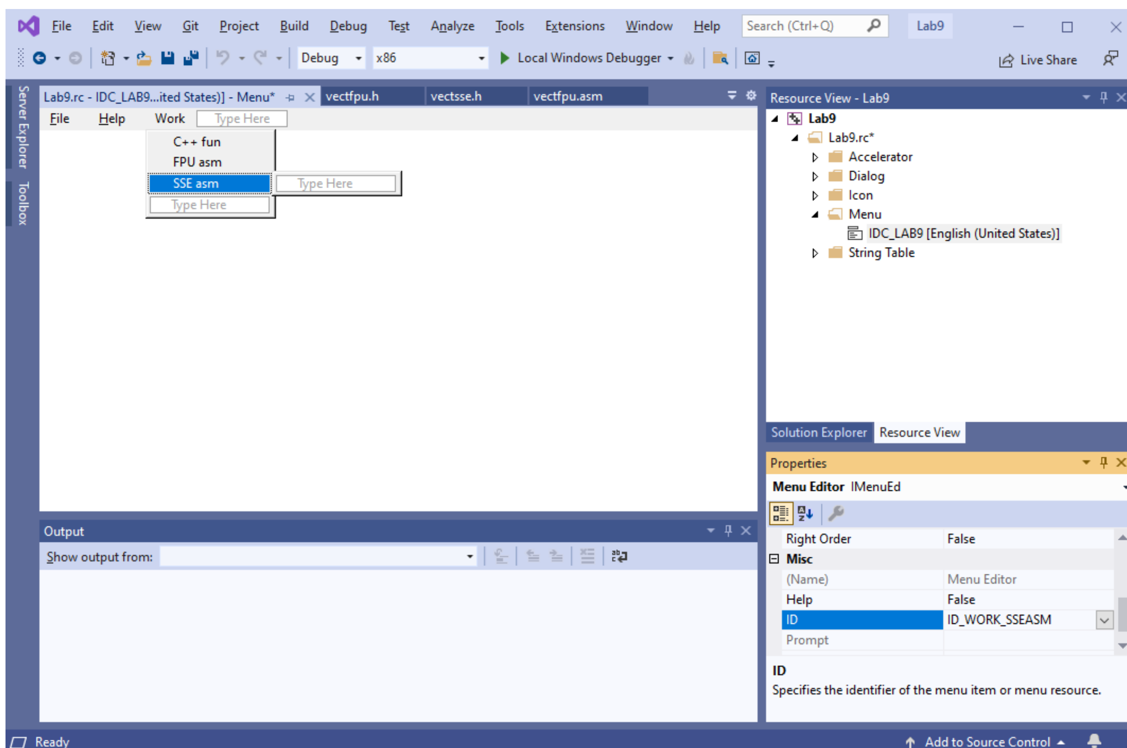


Рис. 9.10. Інтерфейс редагування головного меню програми

Замість “Type here” можна вписувати потрібні пункти меню та групи пунктів. Для кожного пункту меню редактор ресурсів автоматично генерує константу – ID пункту меню і записує її у файл Resource.h. Ці ID будуть потім нами використовуватися для програмування реакції на вибір пунктів меню.

Для перевірки можна виконати “Start Debugging” – наша програма з'явиться з потрібним меню, хоча і без будь-якої змістовної реакції на вибір пунктів меню.

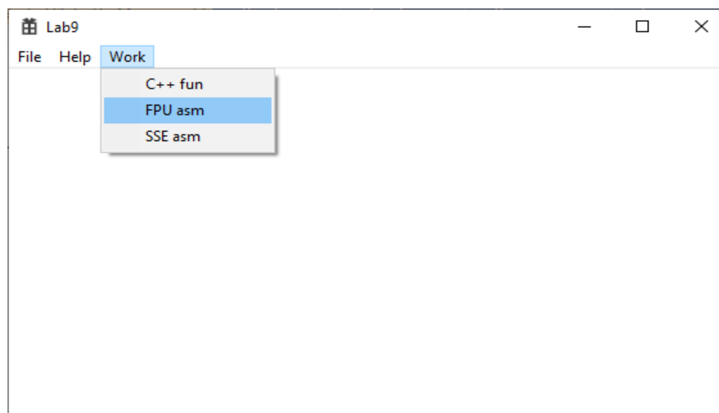


Рис. 9.11. Демонстрація новостворених пунктів меню

Зауваження. Згідно стандарту інтерфейсу користувача пункт Help має бути праворуч усіх інших груп меню. Переставте його у редакторі ресурсів.

Програмування обробників повідомлень меню

Для того щоб при виборі пунктів меню програма виконувала потрібні дії треба вписати програмний код обробників повідомлень у Callback функцію головного вікна. Функція головного вікна у шаблоні має ім’я WndProc і побудована на основі оператора switch(message)

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
            case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
            case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
            }
        }
```

```
        break;
    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code that uses hdc here...
            EndPaint(hWnd, &ps);
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Ця функція обробляє повідомлення, які надходять нашій програмі від Windows. Операційна система стежить, зокрема, і за діями користувача, і при виборі користувачем якогось пункту меню нашій програмі надсилається повідомлення WM_COMMAND. Одним з параметрів цього повідомлення є ID цього пункту меню.

Таким чином, можна додати у switch(message) потрібну кількість case з ID пунктів меню. Кожний case буде позначати обробник повідомлення у вигляді відповідної функції. Фрагмент функції WndProc з такими доробками наведений нижче

```
// попереднє оголошення функцій-обробників меню
void myWorkC(HWND hWnd);
void myWorkFPU(HWND hWnd);
void myWorkSSE(HWND hWnd);
. . .
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
            case ID_WORK_C:          //пункт меню "C++ fun"
                myWorkC(hWnd);
```

```
                break;
            case ID_WORK_FPUASM:     //пункт меню "FPU asm"
                myWorkFPU(hWnd);
                break;
            case ID_WORK_SSEASM:     //пункт меню "SSE asm"
                myWorkSSE(hWnd);
                break;
    . . .
}
```

```
void myWorkC(HWND hWnd)
{
    MessageBox(hWnd, L"Поки що не реалізовано", L"Заглушка", MB_OK);
}

void myWorkFPU(HWND hWnd)
{
    //сюди ми ще допишемо потрібний код
}

void myWorkSSE(HWND hWnd)
{
    //сюди ми ще допишемо потрібний код
}
```

Функції-обробники повідомлень меню поки що зробимо порожніми – як “заглушки”. А можна використати, наприклад, вікно MessageBox, щоб впродовж роботи програми переконатися, що при виборі меню викликається саме той обробник, що треба (рис. 9.12).

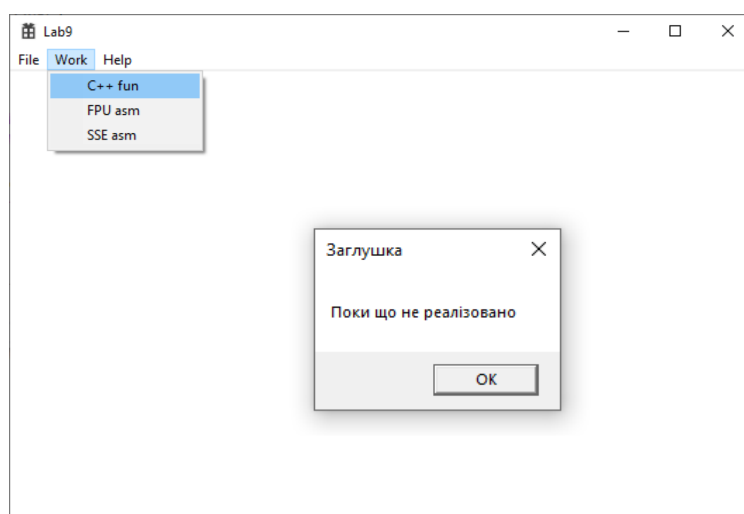


Рис. 9.12. Перевірка меню

Таким чином, вибір пунктів меню, загалом, організовано. Зроблено шаблон-скелет програми. Далі потрібно у тіло функцій обробників вставляти програмний код відповідно варіанту завдання. Зокрема, виклики процедур, які будуть експортуватися з модулів на асемблері.

Виклик процедур з модулів на асемблері

Продовжимо розробку програми методом "з гори до низу". Тепер почнемо розробляти процедури у відповідних модулях. Розпочнемо з інтерфейсу модулів та процедур – тобто узгодження того, як вони будуть викликатися, які параметри їм передавати і як зчитувати результат їхньої роботи.

Уявимо, що в тілі обробника меню "FPU asm" ми маємо викликати процедуру **MyDotProduct_FPU** з модуля **vectfpu.asm**. Зокрема, у якості параметрів передавати два масиви float A[] та B[] і зчитувати результат типу float. Задля універсальності процедури можна також передбачити для неї ще один параметр – кількість елементів масивів. Можна це організувати, наприклад, так

```
float arrayA[1000];
float arrayB[1000];
```

```
void myWorkFPU(HWND hWnd)
{
    //. . .     ще якійсь код
    float res = MyDotProduct_FPU(arrayA, arrayB, 1000);
    //. . .     ще якійсь код
}
```

Процедура **MyDotProduct_FPU** у цьому варіанті має два параметри типу вказівників (float *) і один цілочисельний параметр, може бути, наприклад, типу int, long або якимось ще. Крім того, ця процедура також має повертати значення результату типу float. На асемблері цілком можливо реалізувати таку процедуру, якщо врахувати, як треба запрограмувати повернення результату згідно конвенції _cdecl для функцій C/C++.

А можна організувати код трохи по-іншому – щоб процедура повертала результат через параметр-вказівник:

```
void myWorkFPU(HWND hWnd)
{
    float res = 0;
```

```
    //. . .     ще якійсь код
    MyDotProduct_FPU(&res, arrayA, arrayB, 1000);
    //. . .     ще якійсь код
}
```

Припустимо, що ми оберемо саме такий варіант.

Наступним кроком буде визначення інтерфейсного програмного коду модуля на асемблері. Розглянемо це на прикладі модуля **vectfpu**.

У файлі vectfpu.h запишемо заголовок (прототип) процедури

```
extern "C"
{
      void MyDotProduct_FPU(float* res, float* pa, float* pb, long n);
}
```

Інтерфейс модуля визначено, тепер можна розпочати писати код у файлі vectfpu.asm. Наприклад, так

```
.386
.model flat, C

.code
MyDotProduct_FPU proc res:DWORD, pa:DWORD, pb:DWORD, n:DWORD

  ; а тут буде якійсь код цієї процедури

  ret
MyDotProduct_FPU endp

end
```

Зверніть увагу на параметри процедури – у файлі vectfpu.asm усі вони типу DWORD, незважаючи на те, що у файлі vectfpu.h вони типізовані як float* та long. Пояснення: у хедері вони типізовані для виклику з коду на C/C++, а на асемблері таких типів немає. Асемблер розрізнює типи тільки по кількості бітів. Для 32-бітних програм для моделі flat вказівник – це цілочисельне 32-бітове значення, тобто DWORD.

Таким чином ми підготували шаблон-скелет модуля. У тіло процедури маємо записати асемблерний код відповідно завданню.

Так само можна створювати інший модуль, зокрема, **vectsse**. Його заголовок (файл vectsse.h) повністю подібний файлу vectfpu.h. А у файлі vectsse.asm мають бути деякі відмінності, оскільки планується використати команди SSE.

```
.686
.xmm
.model flat, C

.code
MyDotProduct_SSE proc res:DWORD, pa:DWORD, pb:DWORD, n:DWORD

  ; а тут буде якійсь код цієї процедури

  ret
MyDotProduct_SSE endp

end
```

Як використати такі програмні модулі? Щоб можна було викликати процедури у головному файлі Lab9.cpp, треба записати у верхніх рядках тексту Lab9.cpp директиви #include заголовків потрібних модулів

```
#include "framework.h"
#include "Lab9.h"
#include "vectfpu.h"
#include "vectsse.h"
```

```
void myWorkFPU(HWND hWnd)
{
    float res = 0;
    //. . .     буде ще якійсь код
    MyDotProduct_FPU(&res, arrayA, arrayB, 1000);
    //. . .     буде ще якійсь код
}
void myWorkSSE(HWND hWnd)
{
    float res = 0;
    //. . .     буде ще якійсь код
    MyDotProduct_SSE(&res, arrayA, arrayB, 1000);
    //. . .     буде ще якійсь код
}
```

Тепер можна перевірити коректність програмного коду навіть поки що без повного коду обробників меню та з майже порожнім кодом асемблерних процедур. Виконайте “Start debugging” – Visual Studio скомпілює усі файли і, якщо не буде помилок, на екрані з’явиться вікно програми. Змістовні функції ще не працюють, як треба, але шаблон, скелет програми на цьому етапі буде вже налагоджено. Можна рухатися далі. А для цього необхідно розглянути деякі відомості з предметної галузі призначення програми.

Програмування скалярного добутку на основі команд x87 FPU

Для програмування обчислення скалярного добутку можна використати такі команди x87 FPU:

```
FADD операнд
```

– виконується додавання операнду до ST(0) і запис результату у ST(0).

```
FMUL операнд
```

– виконується множення операнду на ST(0) і запис результату у ST(0).

Примітка. Є також інші різновиди цих команд (дивіться у документації Intel® 64 and IA-32 Architectures. Software Developer's Manual).

А далі розглянемо приклад обчислення суми двох пар добутків

$$Res = A \times B + C \times D$$

```
.data
  valA dq 5.6    ;A
  valB dq 2.4    ;B
  valC dq 3.8    ;C
  valD dq 10.3   ;D
  Res dq ?

.code
main:
  fld valA
  fmul valB
  fld valC
  fmul valD
  faddp st(1), st(0)
  fstp res    ;Res=ST(0)
```

Для виконання потрібних операцій командами FPU необхідно завантажувати дані у стек FPU. Вершина стеку, яка зветься ST(0), спочатку має адресу 0 (вказує на регістр R0 стеку з восьми регістрів R0-R7). Команда FLD завантажує одне число у стек – спочатку зменшується на одиницю адреса вершини стеку ST(0), а потім у ST(0) записується значення операнду.

Оскільки для виконання завдання даної лабораторною потрібно додавати багато пар добутків, то для їхнього зберігання рекомендується створити масиви, а для обчислення необхідно запрограмувати цикл, у якому будуть виконуватися команди множення та додавання чисел з плаваючою точкою.

Програмування скалярного добутку на основі команд SSE

Векторні команди SSE

Ці команди виконують однакову операцію одразу над чотирма парами чисел в 32-бітовому форматі з плаваючою точкою, наприклад

```
addps A, B     ;додавання чотирьох чисел
subps A, B     ;додавання чотирьох чисел
mulps A, B     ;множення чотирьох чисел
divps A, B     ;ділення
rcpps A, B     ;зворотна величина
sqrtps A, B    ;квадратний корінь
rsqrtps A, B   ;1/квадратний корінь
maxps          ;максимум
minps          ;мінімум
```

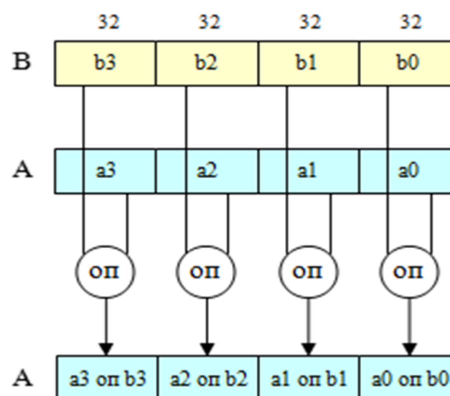


Рис. 9.13. Приклади векторних команд SSE

Для обчислення скалярного добутку *n*-вимірних векторів можна скористатися командою **`mulps`**, яка перемножує одночасно чотири пари чисел – елементів векторів. Для зручності організації циклу обчислень потрібно, щоб *n* було кратним 4.

Накопичення сум добутків можна виконувати командою **`addps`**. Ця команда додає одночасно чотири пари чисел і зберігає чотири результати у відповідні частини якогось регістру XMM. Так накопичуються суми добутків у вигляді чотирьох чисел у регістрі, наприклад, XMM0.

Горизонтальне додавання

Для отримання кінцевого результату скалярного добутку потрібно обчислити суму чотирьох елементів регістру XMM0, тобто виконати так зване "горизонтальне" додавання

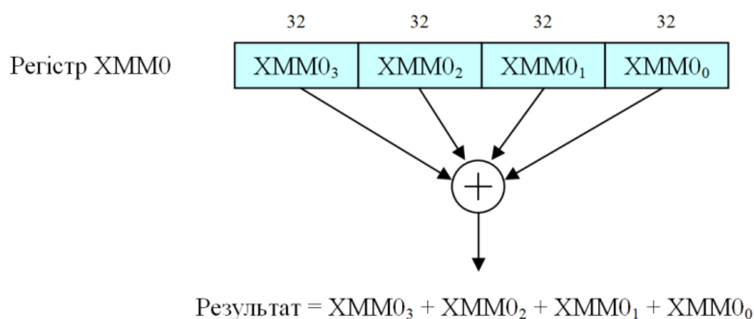


Рис. 9.14. "Горизонтальне" додавання елементів регістру

Знаходження суми усіх чвертинок регістру XMM0 можна виконати двома командами **haddps**:

```
haddps xmm0, xmm0
haddps xmm0, xmm0
```

Одна команда **haddps** працює так:

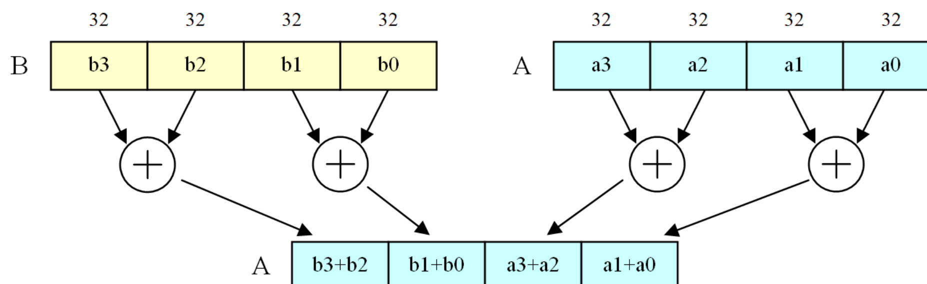


Рис. 9.15 Схема виконання команди **haddps A,B**

Вирівнювання даних

Деякі команди SSE, які зчитують дані з пам'яті, коректно працюють лише у випадку, якщо ці дані у пам'яті вирівнені на межу 16 байтів. Те саме стосується і запису даних у пам'ять за деякою адресою. Іншими словами, адреса джерела та адреса призначення повинна бути кратною 16. Наприклад, командою

```
movaps xmm1, [ebx]
```

записується квадрослово (16 байтів) у регістр XMM1 з пам'яті за адресою, яка міститься у регістрі EBX – ця адреса повинна бути кратною 16, інакше виконання програми буде аварійно завершене.

На відміну цього, командою

```
movups xmm1, [ebx]
```

можна зробити те саме і для невирівнених даних, проте команда **movups** працює суттєво повільніше, аніж **movaps**. Це обов'язково потрібно враховувати. Якщо продедура на асемблері обробляє дані, запрограмовані у модулях на C++, то ці дані повинні бути вирівнені. Запрограмувати на C++ створення статичного масиву, адреса якого кратна 16, можна так:

```
__declspec( align(16) ) float a[1000];
```

Як запрограмувати контроль часу виконання потрібних функцій?

Одним з варіантів рішення є використання функції API Windows **GetLocalTime**, наприклад, у такий спосіб:

```
SYSTEMTIME st;
long tst,msec;
GetLocalTime(&st);
tst = 60000*(long)st.wMinute
    + 1000*(long)st.wSecond
    + (long)st.wMilliseconds;
for (long i=0; i<1000000; i++)    //повторюємо мільйон разів
    {
    . . .          //код, для якого потрібно виміряти час
    }
GetLocalTime(&st);
msec = 60000*(long)st.wMinute
    + 1000*(long)st.wSecond
    + (long)st.wMilliseconds - tst;
```

Для коректного виміру часу потрібно встановити таку кількість повторень, щоб час виконання циклу був не менше секунди.

Вивід результатів у вікні MessageBox

У вікні MessageBox можна показати будь-який текст, зокрема, числові значення результатів. Текст для вікна MessageBox можна сформувати у буфері рядка символів, наприклад, використавши функцію **sprintf_s** бібліотеки **stdio**. А для проектів з Unicode (які натепер Visual Studio створює за умовчанням) можна порекомендувати такий спосіб:

```
#include <stdio.h>

char buf[128];
WCHAR text[128];
sprintf_s(buf, sizeof(buf), "Результат = %f, час: %d мсек", res, msec);
MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, buf, strlen(buf)+1, text, 128);

MessageBox(hWnd, text, L"Результат (вказати результат чого)", MB_OK);
```

Варіанти завдань та основні вимоги

1. У середовищі MS Visual Studio створіть новий проект C++ з ім'ям **Lab9**.

2. Запрограмуйте у модулях на асемблері процедури обчислення скалярного добутку. Процедури двох видів:

`MyDotProduct_FPU(&res, A, B, N)` на основі команд x87 FPU

`MyDotProduct_SSE(&res, A, B, N)` на основі команд SSE

Також треба запрограмувати вже мовою C/C++ третю процедуру (функцію) для обчислення скалярного добутку

```
MyDotProduct(&res, A, B, N);
```

У головному файлі Lab9 мають бути виклики трьох функцій. Вибір – через меню.

3. Кількість елементів векторів A та B має бути

$$N=480 \times \text{(номер студента у журналі)}$$

4. Запрограмуйте знаходження часу виконання для кожного з варіантів обчислення скалярного добутку.

5. Програма повинна виводити числове значення скалярного добутку та час виконання у вікні MessageBox – окремо для кожного з варіантів реалізації.

6. Отримайте дизасемблерний код процедури MyDotProduct на C++ і порівняйте з кодом процедури на асемблері MyDotProduct_FPU. Проаналізуйте відмінності програмних кодів, якщо вони є.

Можливі бонуси

Оцінка може бути суттєво підвищена, якщо окрім модуля SSE студент запрограмує на асемблері також модуль скалярного добутку на основі команд AVX.

Зміст звіту

1. Титульний лист
2. Завдання
3. Роздруківка вихідного тексту програми:
   - вихідних текстів файлів модулів: *.asm та *.h – повністю
   - головного файлу lab9.cpp – частково, тільки текст, який був доданий для виконання завдання
   - дизасемблерний код функції **MyDotProduct** на C++
4. Скріншоти виконання програми
5. Аналіз, коментар результатів та вихідного тексту
6. Висновки

Контрольні запитання

1. Як працюють команди SSE?
2. Що таке "горизонтальне" додавання і яке тоді додавання "вертикальне"?
3. Що таке вирівнювання даних і як воно вказується?
4. Що таке стек даних FPU?
5. Що потрібно вказати, щоб успішно компілювався вихідний текст на асемблері з використанням команд SSE?
6. Як можна запрограмувати виміри часу виконання потрібних операцій?
7. Які Ви знаєте SIMD команди?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Базова

1. Intel® 64 and IA-32 Architectures Software Developer Manuals [Online] Available from: https://www.intel.com/content/www/us/en/ developer/ articles/ technical/intel-sdm.html
2. Microsoft. Creating and Managing Visual C++ Projects [Online] Available from: https://msdn.microsoft.com/en-us/library/4457htyc.aspx
3. Microsoft. Directives Reference [Online] Available from: https://learn.microsoft.com/en-us/cpp/assembler/masm/directives-reference?view=msvc-170
4. Microsoft Macro Assembler reference [Online] Available from: https://learn.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference?view=msvc-170

Додаткова

5. Лекції "Системне програмування" [Online] Available from: https://learn.ztu.edu.ua/mod/book/view.php?id=27881
6. Ставицький О.В. Операційні системи та системне програмування [Online] Available from: https://ela.kpi.ua/bitstream/123456789/26025/1/ practopersyst.PDF
7. Flat assembler [Online] Available from: https://flatassembler.net/index.php
8. Intel® 64 and IA-32 Architectures Optimization Reference Manual [Online] Available from: https://cdrdv2.intel.com/v1/dl/getContent/671488
9. NASM [Online] Available from: https://www.nasm.us/