**Лабораторна робота №3**

**Створення модульних проектів на асемблері у середовищі Visual Studio 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 MS Visual Studio проект з ім'ям **Lab3**.
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 **main3.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 виконуємий 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: $a_{n-1}\ a_{n-2}\ \dots\ a_1\ a_0$ .

Числове значення $A$ можна знайти по формулі:

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

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 (1), використовується для представлення $n$-бітових позитивних чисел (без знаку).

Додатковий код

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

$$[A]_{дк} = A \quad (якщо\ A \geq 0) \qquad (2)$$
$$= r^n - |A| \quad (якщо\ A < 0)$$

тут вважається, 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 додаткового коду дорівнює 1:

$$\begin{array}{rccccccc} & 1 & 0 & 0 & . & . & 0 & 0 \quad (2^n) \\ - & & & & & & & \\ & a_{n-2} & a_{n-3} & . & . & . & a_1 & a_0 \quad (|A|) \\ \hline [A<0]_{дк} = & 1 & d_{n-2} & d_{n-3} & . & . & d_1 & d_0 \end{array}$$

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

$$[A>=0]_{дк} = 0 \quad a_{n-2} \quad a_{n-3} \ . \ . \ . \ a_1 \quad a_0$$

Старший розряд називають **знаковим** – його цифра прямо вказує на знак числа (якщо не було переповнення розрядної сітки після виконання деякої операції).

Для перетворення у додатковий 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.

При виконанні операцій додавання та віднімання чисел знакові розряди обробляються так само, як інші розряди.

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 додатковий код.

Формати з плаваючою точкою

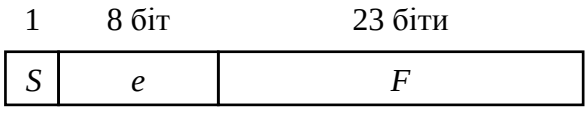
Математичний опис двійкового числа у форматі з плаваючою точкою:

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

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 стандарту IEEE 754.

Одинарний 32-бітовий формат з плаваючою точкою



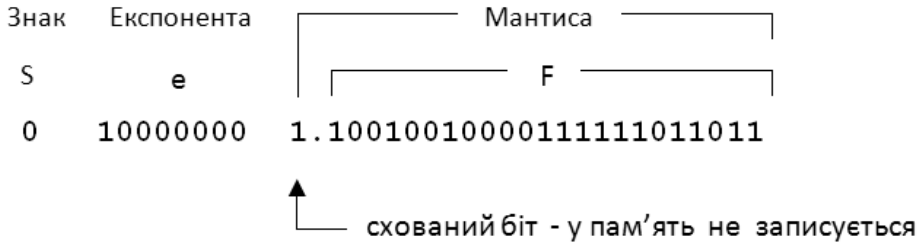

Експонента записується у зміщеному коді: $e = E+127$.
Діапазон для експоненти $E$: від $E_{min}$ = -126 до $E_{max}$=+127. Виходячи з цього, можна оцінити діапазон представлення чисел: ±(від $2^{-126}$ до $2^{+127}$).

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

**3.14159265358979323846264338327...**

у двійковому 32-бітовому форматі з плаваючою точкою. Отримаємо наступне:




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

| 1 | 11 | 52 |
|---|---|---|
| *S* | *e* | *F* |

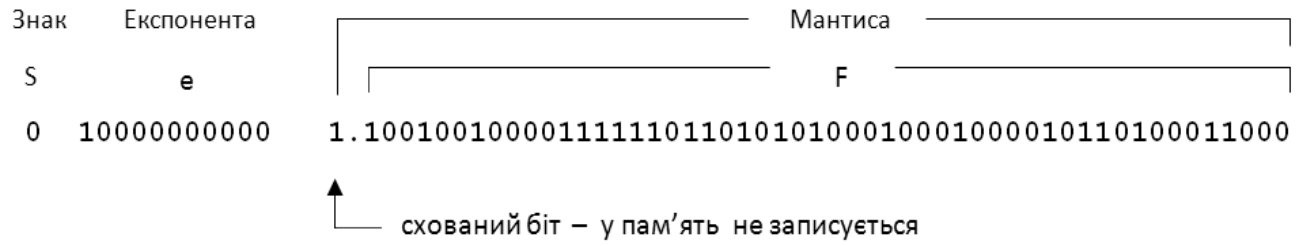
Кодоване значення експоненти: $e = E + 1023$. Експонента $E$ від $E_{min}$ = -1022 до $E_{max}$=+1023, тому діапазон представлення чисел: ±(від $2^{-1022}$ до $2^{+1023}$).

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 число

**3.14159265358979323846264338327 95**

кодується наступним чином:




У пам'яті записується вісім байтів:

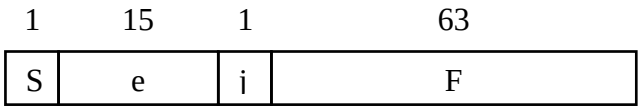
```
Адреса + 7    01000000
Адреса + 6    00001001
Адреса + 5    00100001
Адреса + 4    11111011
Адреса + 3    01010100
Адреса + 2    01000100
Адреса + 1    00101101
Адреса числа  00011000
```

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




Кодоване значення експоненти: e = $E$ + 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 мантиса записується інакше, аніж для 32-, та 64-бітових форматів, і має наступний вигляд:

$M = j.F$,

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.

**Порядок виконання роботи та 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 **Lab3**. Як створювати подібний проект програми на асемблері – про це докладно написано у поясненнях до попередньої лабораторної роботи №2.

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**, 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 Str_HEX 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
```

Тут 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 програмах попередніх робіт №1, 2.

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:

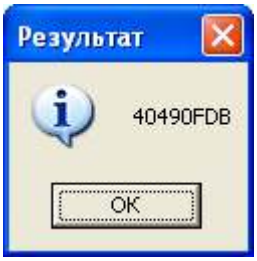



Рис. 1. Шістнадцятковий код 32-бітового числа

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 8-бітних перемінних, DW для 16-бітних, DD для 32-бітних, DQ для 64-бітних, DT для 80-бітних. При створенні кожної перемінної ініціалізувати її потрібним значенням.

**Варіанти завдання**

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 відповідно наведеній нижче таблиці.

| 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 | |
| | -X | шістнадцятковий код | |
| Число у 32-бітовому форматі з плаваючою точкою | X.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | -Y.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | X.X | шістнадцятковий код | |
| Число у 64-бітовому форматі з плаваючою точкою | X.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | -Y.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | X.X | шістнадцятковий код | |
| Число у 80-бітовому форматі з плаваючою точкою | X.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | -Y.0 | шістнадцятковий код | двійковий код (вказати знаковий біт, біти експоненти, біти мантиси) |
| | X.X | шістнадцятковий код | |

Примітка. Двійковий код можна отримати без комп’ютера, перетворивши отримані відповідні шістнадцяткові коди. Якщо студент при виконанні лабораторної роботи запрограмує і двійкові коди – 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 питання:**

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 додатковий 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**?
13. Що змінити у тексті module.asm у коді процедури StrHex_MY, щоб замість

```
push offset TextBuf
push offset Value
push ...                ;кількість бітів Value
call StrHex_MY
```

можна було б записати такий код:

```
push offset TextBuf
push offset Value
push type Value         ;кількість байтів Value
call StrHex_MY
```