Міністерство інфраструктури України

Державна служба зв'язку
Одеська національна академія зв'язку ім. О. С. Попова

Кафедра інформаційних технологій

О. Г. Трофименко, Л. Л. Леоненко

Організація обчислень засобами вбудованого у C++ Builder асемблера

Методичні вказівки до практичних і лабораторних робіт з дисциплін «Комп'ютерні технології та програмування», «Мови програмування»

для студентів напрямів бакалаврської підготовки:
«*Автоматизація та комп'ютерно-інтегровані технології*»,
«*Системи технічного захисту інформації*»

Одеса 2011

УДК 004.43

План НМВ 2011 р.

Трофименко О. Г. Організація обчислень засобами асемблера, вбудованого у C++ Builder: метод. вказівки до лаб. та практ. занять. Модуль 2.2 / Трофименко О. Г., Леоненко Л. Л. – Одеса: ОНАЗ ім. О.С. Попова, 2011. – 116 с.

Розглянуто основні засоби застосування асемблера, вбудованого у Borland C++ Builder. Містить теоретичні відомості про програмно-доступні апаратні пристрої комп'ютерів, основні команди асемблера та приклади організації програмних проектів із вбудованим асемблером для кожної з поданих тем. Згідно з навчальними планами у вказівках подано індивідуальні контрольні питання для підготовки до практичних занять, індивідуальні завдання для виконання лабораторних робіт, комплексного завдання та курсової роботи.

Призначено для набуття теоретичних та практичних знань студентами академії, які вивчають дисципліни "Комп'ютерні технології та програмування", "Мови програмування". Буде корисним для студентів при закріпленні лекційного матеріалу та підготовці до практичних і лабораторних занять.

ЗАТВЕРДЖЕНО
методичною радою академії
Протокол № 8 від 11.02.2011 р.

СХВАЛЕНО
на засіданні кафедри
інформаційних технологій
і рекомендовано до друку

Протокол № 7 від 01.02.2011 р.

Здано в набір 20.09.2011 Підписано до друку 28.09.2011
Формат 60x90/16 Зам. № 46
Наклад 300 прим. Обсяг 7,25 друк. арк.
Віддруковано на видавничому устаткуванні фірми RISO
у друкарні редакційно-видавничого центру ОНАЗ ім. О.С. Попова
м. Одеса, вул. Ковалевського, 5
Тел. 720-78-94
ОНАЗ, 2011

ПЕРЕДМОВА

Дисципліна "*Комп'ютерні технології та програмування*" вивчається студентами академії за напрямом бакалаврської підготовки "Автоматизація та комп'ютерно-інтегровані технології", в семестрах: 1.1, 1.2, 1.3, 1.4, 2.1 та 2.2 першого та другого курсу навчання.

Вивчення цієї дисципліни розраховано на шість модулів:

- модуль 1.1. "Основні відомості про персональний комп'ютер та його програмне забезпечення";
- модуль 1.2. "Алгоритмізація обчислень і особливості програмування у C++";
- модуль 1.3. "Програмне опрацювання масивів у C++";
- модуль 1.4. "Програмне опрацювання символьних даних та файлів у C++";
- модуль 2.1. "Організація обчислень засобами асемблера, вбудованого у C++ Builder";
- модуль 2.2. "Діагностика комп'ютерів".

Дисципліна "*Мови програмування*" вивчається студентами академії за напрямом бакалаврської підготовки "Системи технічного захисту інформації" в семестрах 2.1 та 2.2 другого курсу навчання.

Вивчення цієї дисципліни розраховано на два модулі:

- модуль 2.1. "Апаратно-залежні мови програмування";
- модуль 2.2. "Організація обчислень засобами асемблера, вбудованого у C++ Builder".

Програма дисципліни містить курсову роботу: "Розрахунок складних трансцендентних функцій, знаходження екстремумів та обробка масивів засобами асемблера у середовищі C++ Builder".

Модулі 2.1 та 2.2 в обох дисциплінах та курсова робота в дисципліні "Мови програмування" передбачають вивчення основних засобів застосування вбудованого у Borland C++ Builder асемблера, оскільки використання асемблера у програмах C++ дає можливість отримати доступ до найнижчих рівнів керування обладнанням, що є необхідним для організації систем захисту інформації.

Перелік знань та умінь, яких повинен набути студент у процесі вивчення матеріалу модулів 2.1 та 2.2

Знання:

- архітектура мікропроцесора INTEL. Регістри процесора;
- формат мови асемблер. Використання асемблерних вставок у C++ Builder;
- арифметичні команди опрацювання регістрів процесора CPU і співпроцесора FPU;
- логічні операції мови асемблер та команди керування асемблер-програмою;

– команди, які забезпечують взаємодію процесора CPU з співпроцесором FPU.

Вміння:

– організовувати асемблерні вставки у програмах C++ Builder;
– розробляти асемблерний програмний код для виконання арифметичних операцій;
– розробляти асемблерний програмний код з розгалуженнями та циклами;
– організовувати взаємодію процесора CPU з співпроцесором FPU у асемблер-програмі.

Кожна із запропонованих до виконання лабораторних робіт має індивідуальне завдання. Метою завдань є набуття практичних навичок розв'язування інженерних задач з використанням широких можливостей асемблера щодо програмного доступу до апаратних засобів комп'ютера; використання чисельних методів та закріплення знань і вмінь програмувати алгоритмічною мовою C++.

Кожне лабораторне завдання містить 30 індивідуальних варіантів. Студент обирає варіант завдання згідно з його номером у списку групи.

До виконання лабораторної роботи допускається студент, який самостійно підготував протокол лабораторної роботи відповідно до поданих нижче вимог. Після виконання лабораторної роботи студент повинен занести до протоколу результати обчислень та зробити відповідні висновки. Правильність здобутих результатів перевіряє викладач.

Перелік лабораторних робіт

Лабораторна робота № 1. Арифметичні операції над цілими числами у вбудованому в C++ Builder асемблері.

Лабораторна робота № 2. Арифметичні операції над дійсними числами у вбудованому в C++ Builder асемблері.

Лабораторна робота № 3. Опрацювання числових масивів засобами вбудованого у C++ Builder асемблера.

Лабораторна робота № 4. Опрацювання рядків засобами асемблера.

Вимоги щодо оформлення протоколу лабораторної роботи

1. Лабораторні роботи оформлюються в окремому зошиті.
2. Для кожної лабораторної роботи слід записати тему та мету.
3. Для задачі лабораторної роботи слід записати такі розділи:
   а) умову задачі за індивідуальним варіантом;
   б) опис розв'язування задачі на комп'ютері;
   в) результати обчислень на комп'ютері;
   г) аналіз результатів та висновки.
4. Наприкінці роботи треба написати своє прізвище, поставити підпис і дату виконання роботи.

ВСТУП

На фоні широкої популярності мов високого рівня C, Pascal, Basic та ін. мова низького (машинного) рівня – мова асемблер – не втрачає своєї актуальності і продовжує широко використовуватись у розробленні програм, оскільки асемблер є мовою, якою "розмовляє" процесор. Отже, зникнути ця мова може лише разом зі зникненням процесорів. До того ж, асемблер має фундаментальну перевагу перед мовами високого рівня: він є найшвидшим. Більшість програм, які працюють в режимі реального часу, або є написані цілковито на асемблері, або використовують у критичних ділянках коду асемблерні модулі.

Асемблерний код є більш компактним, ніж його аналог мовою високого рівня. У цьому легко переконатись, порівнявши дизасембльовані лістинги однієї й тієї самої програми, написаної асемблером та мовою високого рівня.

Можна розробляти як окремі модулі на асемблері, так і долучати їх до програм, написаних мовами високого рівня. Також можна використовувати широкі можливості вбудованого асемблера, передбачені в багатьох мовах високого рівня. Вбудований асемблер дозволяє досягти максимального ефекту при оптимізації математичних виразів, програмних циклів опрацювання масивів. Провідні фірми-виробники, такі як Microsoft та Borland, невпинно вдосконалюють вбудований асемблер.

Короткі швидкі програми мовою асемблер застосовуються там, де розміри коду та його швидкодія є критичними параметрами. Сферами застосовування таких програм є системи реального часу, системні утиліти та програми, а також драйвери пристроїв. Програми на асемблері керують як периферійними пристроями персонального комп'ютера (ПК), так і нестандартними пристроями, приєднаними до ПК.

У посібнику розглядаються основні засоби застосування вбудованого у Borland C++ Builder асемблера. Мова C++ є дуже популярна, що пояснюється високою ефективністю об'єктних кодів C++-програм як за швидкодією, так і за обсягом пам'яті. Слід нагадати, що мова C бере свій початок від мови BCPL, яка була мовою-макроасемблером. Для більшості програм компілятори C++ генерують компактний та швидкий машинний код, а розумне застосовування вбудованого у C++ асемблера надає можливість збільшити швидкодію програм і отримати доступ до найнижчих рівнів керування обладнанням.

Посібник призначається для підготовки до лабораторних робіт з дисциплін "Комп'ютерні технології та програмування" та "Мови програмування". Передбачено можливість виконання чотирьох лабораторних робіт. Перед кожною лабораторною роботою розміщено докладні теоретичні відомості й наведено приклади програм з аналізом програмного коду.

Тема № 1

АПАРАТНІ ЗАСОБИ КОМП'ЮТЕРІВ

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Терміном *"архітектура комп'ютера"* називають *систему принципів, покладених в основу проектування комп'ютера* певного типу. Більш вузьке значення цього терміну пов'язане з *системою основних апаратних пристроїв (hardware) комп'ютера та засобів організації взаємодії між цими пристроями*. Оскільки таку взаємодію забезпечують (принаймні частково) програми операційної системи, вивчення архітектури пов'язане з вивченням однієї (чи кількох основних) операційних систем, призначених для комп'ютерів певного класу.

Ми вивчатимемо архітектуру персональних комп'ютерів (ПК), сумісних з типом IBM PC, та найбільш уживану на цих комп'ютерах операційну систему (ОС) сімейства Windows.

1.1. ПРОЦЕСОР IBM PC-СУМІСНОГО КОМП'ЮТЕРА

Мікропроцесор (МП чи CPU – central processor unit) є основним електронним чіпом (мікросхемою) на системній платі, що по суті є мозком комп'ютера, який забезпечує виконання обчислень та керування іншими його пристроями. Процесор має свою внутрішню пам'ять, організовану у вигляді так званих *регістрів*. Різні типи та серії процесорів (Intel Pentium, Intel Celeron, Intel Xeon, Intel Core, Intel Itanium, AMD Sempron, AMD Athlon, AMD Phenom) відрізняються, зокрема, *розрядністю* (розмірами своїх регістрів), тактовою частотою та низкою інших особливостей, які і визначають основні можливості ПК, як от: амплітудою робочої температури, габаритами, споживанням енергії тощо.

Розміром чи *розрядністю* регістра називають максимальну кількість біт інформації, яку можна в них розмістити. Перші комп'ютери IBM PC були обладнані 16-розрядними процесорами i8086 чи i8088 фірми Intel. Це було перше покоління "сім'ї" процесорів, яке дістало назву x86. Наступне покоління у цій сім'ї – процесори i80286 – також було 16-розрядним. Покоління i80386, i80486 та перші Pentium'и (i80586) мали 32-розрядні процесори. Сучасні моделі мають 64-бітну розрядність.

Нагадаємо, що кожен з бітів може зберігати одне з двох можливих значень: 0 чи 1. За допомогою одного байта (8-ми бітів) можна закодувати ціле додатне число в діапазоні від 0 до 265 (всього 256 комбінацій, оскільки $2^8 = 256$), двох байтів – число від 0 до 65 535 ($2^{16} = 65\,536$), чотирьох байтів – число від 0 до 4 294 967 295 ($2^{32} = 4\,294\,967\,296$). Саме цим зумовлені розмірності цілочисельних типів даних у різних мовах програмування.

Крім розрядності процесора, на його продуктивність суттєво впливає тактова частота його роботи. *Тактова частота* – кількість операцій, виконуваних

МП за секунду.[1] Частота вимірюється в герцах (гц), а тактова частота сучасних ПК в Ггц. МП, навіть однакового типу, можуть мати різні частоти – і чим частота вища, тим краща продуктивність ПК.

Окрім названих, ще одною характеристикою швидкодії комп'ютера є розрядність *обміну даними* між процесором та іншими пристроями. Деякі комп'ютери з 64-розрядними процесорами передбачають лише 32-розрядний обмін даними (їх ще називають комп'ютерами з 32-розрядною шиною обміну). Швидкодія цих комп'ютерів нижча, ніж у машин з такою само тактовою частотою, але з 64-розрядною шиною.

Процесори зазвичай містять від декількох до сотень регістрів різного функціонального призначення. *Регістр процесора* – комірка швидкодіючої внутрішньої пам'яті процесора, яка використовується для тимчасового зберігання операндів, з якими безпосередньо проводяться обчислення, а також часто використовуваних даних з метою швидкого доступу до них. Крім того, в регістрах зберігається і додаткова інформація, потрібна процесору для функціонування (зокрема, параметри поточного стану процесора, адреса команди, яка буде виконуватись у наступний момент тощо).

З точки зору архітектури комп'ютера, термін *регістр* вживають лише для тих регістрів, які доступні програмісту в рамках документованої програмної моделі процесора. Більш точно такі регістри називають архітектурними. Наприклад, архітектура x86 визначає лише вісім 32-розрядних регістрів загального призначення, але процесор фактично містить набагато більше реальних апаратних регістрів. Така надлишковість потрібна для реалізації деяких мікроархітектурних оптимізацій швидкодії процесора.

Усі процесори сімейства x86 (а також подібних чи сумісних сімейств) мають одну й ту ж саму базову систему регістрів; найменування та призначення цих регістрів зберігається незалежно від зміни розрядності та інших характеристик процесорів. У табл. 1.1 подано перелік архітектурних (доступних з програм) 16-розрядних та відповідних 32-розрядних регістрів.[2]

Таблиця 1.1. **Базова система регістрів процесорів сімейства x86**

Ім'я 16-розрядного регістра	**Ім'я 32-розрядного регістра**	**Англійська назва – "псевдонім"**	**Основне призначення**
1	*2*	*3*	*4*
Регістри загального призначення			
AX	EAX	Accumulator	Регістр для виконання арифметичних операцій
BX	EBX	Base	Використовується для адресації за базою
CX	ECX	Counter	Лічильник кроків циклу

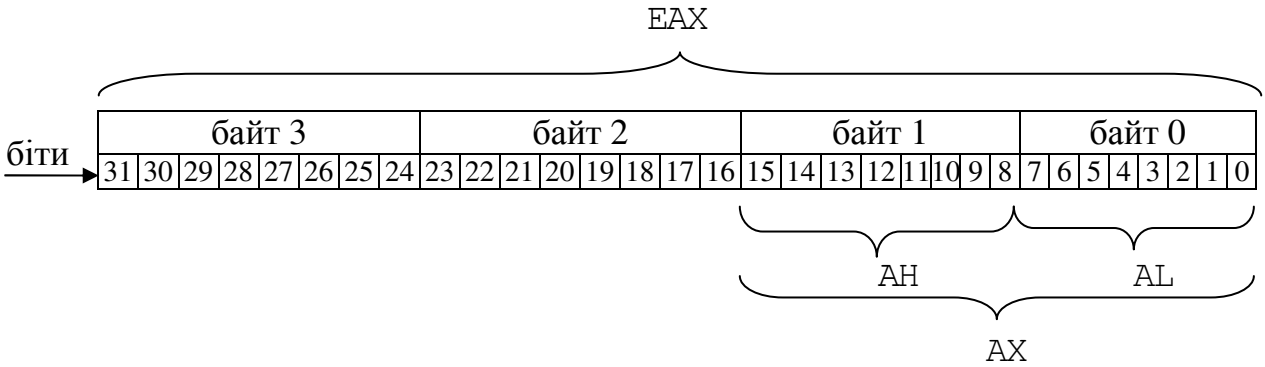
[1] Існують команди мови асемблера, які процесор виконує протягом кількох тактів, проте жодну команду він не може виконати швидше, ніж за 1 такт.

[2] Імена регістрів можна записувати як великими літерами, так і малими, отже, обидві форми запису – EAX та eax – є рівноправними у текстах мовою асемблера.

Закінчення табл. 1.1

1	*2*	*3*	*4*
DX	EDX	Data	Зберігає “довгі” результати операцій (які не вміщуються в AX)
SI	ESI	Source Index	Індекс поточного елемента області, з якої пересилаються дані
DI	EDI	Destination Index	Індекс поточного елемента області, куди пересилаються дані
BP	EBP	Base Pointer	Адреса бази стека
SP	ESP	Stack Pointer	Адреса вершини стека
Сегментні регістри			
CS		Code Segment	Адреса сегмента команд
SS		Stack Segment	Адреса сегмента стека
DS		Data Segment	Адреса сегмента даних
ES, FS, GS		Extension Segments	Адреси “додаткових” сегментів
Регістр прапорців			
FLAGS	EFLAGS	Flags	Містить “прапорці” – індикатори настання певних подій (приміром: чи не відбулось *переповнення* регістра внаслідок виконання попередньої арифметичної операції; чи є результат попередньої операції *нулем* тощо)
Регістр – вказівник команд			
IP	EIP	Instruction Pointer	Містить адресу команди, яку процесор виконує на поточному кроці своєї роботи

Порівняно з базовими 16-розрядними регістрами, відповідні 32-розрядні регістри називають “розширеними” (extended). Тобто перша літера “E” на початку імен 32-розрядних регістрів – EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, EFLAGS та EIP – тлумачиться як скорочення від “extended”. Суть “розширення” пояснюється наступним прикладом: 16-розрядний регістр AX є *частиною* (молодші два байти) його повного 32-бітного регістра EAX:



Таким чином, у 32-розрядних процесорах AX вже не є *фізичною* коміркою пам'яті. Проте, збережено можливість *звертатись до* AX *у командах*, які виконує процесор. Отже, якщо програма, написана для 16-розрядного процесора, містила звертання до AX, вона може без будь-яких змін виконуватись 32-розрядним процесором.

З поданого рисунка ви бачите, що два байти регістра AX (два молодші байти EAX) мають "власні імена" – AH та AL. За цими іменами до них також можна звертатись у командах процесора. До AH та AL також застосовують термін *регістр*, хоча, зрозуміло, вони є не фізичними, а "*логічними*" регістрами.

Структуру, аналогічну структурі регістра EAX, мають ще три регістри – EBX, ECX, EDX. Отже, існують:

– "логічні" регістри BH, CH, DH (літера "H" – від англ. High);

– "логічні" регістри BL, CL, DL ("L" – від англ. Low).

Перші вісім регістрів, поданних у табл. 1.1 (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP), називають *регістрами загального призначення*. Вони можуть без обмежень використовуватись у програмах для тимчасового зберігання опрацьовуваних даних, аргументів чи результатів операцій. Назви-"псевдоніми" регістрів (подані у третій колонці табл. 1.1) походять від того, що деякі команди застосовують відповідний регістр в особливий спосіб. Приміром, акумулятор (EAX) вживається для "накопичення" результатів арифметичних операндів, – скажімо, команда `imul ECX` виконує множення числа з регістра ECX на поточний вміст регістра EAX, і результат зберігається в EAX.[3] Регістр-лічильник (ECX) працює як лічильник у циклах і операціях з рядками, а регістр-база (EBX) використовується для так званої базової адресації.

Індексні регістри ESI (індекс джерела) та EDI (індекс приймача) застосовуються для так званих ланцюжкових операцій пересилання даних, тобто операцій послідовного опрацювання ланцюжків елементів, кожен з яких може мати розмірність 32, 16 чи 8 бітів. Регістр ESI в таких операціях містить поточну адресу елемента в ланцюжку-джерелі, а регістр EDI – поточну адресу в ланцюжку-приймачі.

Регістри ESP та EBP, зазвичай, пов'язані зі стеком. Використання цих регістрів для арифметичних чи логічних команд можливе, але небажане. *Стек* – це область пам'яті, організована для тимчасового зберігання даних за принципом: "першим зайшов, останнім вийшов", тобто елементи з нього виймаються в порядку, зворотному до їхнього записування. На вершині стека зберігатиметься останній з записаних елементів. "Наочною" моделлю стека може служити вертикальний циліндр з підпружиненим дном. Через верхній відкритий кінець циліндра в нього можна "завантажувати", чи, як то кажуть, "заштовхувати" елементи. Загальноприйняті англійські терміни в цьому плані дуже яскраві, операція записування елемента в стек позначається PUSH, в перекладі "заштовхнути,

[3] Якщо ж цей результат не вміщується в акумуляторі, його старша частина розміщується у регістрі даних (EDX), а молодша – в EAX.

запхнути". Новий записуваний елемент проштовхує елементи, які записано в стек раніше, на одну позицію вниз. При зчитуванні елементів зі стека вони наче "виштовхуються" догори, по-англійськи POP ("вистрілюють"). У наступних розділах буде детальніше розглянуто роботу команд PUSH та POP, які коректно записують та зчитують дані з вершини стека без надання цим даним жодних імен. Регістр вказівника стека ESP (Stack Pointer) завжди вказує на вершину стека в поточному сегменті стека (SS), тобто на зсув останнього елемента в стеку відносно початку стека. Регістр вказівника бази кадра стека EBP (Base Pointer) призначений для організації довільного доступу до даних усередині стека. Крім тимчасового зберігання опрацьовуваних даних, стек традиційно широко використовується для зберігання вмісту регістрів перед викликом підпрограми. Первинний стан регістрів відновлюється зі стека після повернення з підпрограми. Іншим поширеним прийомом використання стека є передавання підпрограмі необхідних їй параметрів через стек.

Насправді, у 32-розрядних процесорах в якості базового чи індексного може виступати будь-який регістр, тобто при програмуванні зберігати операнди команд можна в більшості регістрів, причому практично в будь-яких поєднаннях. Проте, завжди слід пам'ятати, що деякі команди працюють лише з певними регістрами. Знання особливостей використання регістрів машинними командами дозволяє, за потреби, економити пам'ять, займану кодом програми, і програмувати більш ефективні алгоритми.

При переході з 32-розрядних версій на 64-розрядні префікс "E" у всіх іменах регістрів було змінено префіксом "R" (скажімо, RAX – спадкоємець EAX, RBX – нащадок EBX, RSI замінює ESI і т. п.). Крім того, з'явилося вісім нових регістрів загального призначення (R8 – R15). Тобто, сучасні 64-розрядні процесори x86_64 мають 16 цілочисельних 64-бітних регістрів загального призначення (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 – R15). Звичайно, програмно доступними лишилися і всі попередні версії регістрів (EAX, AX, AL, AH та ін.).

Слід, проте, зазначити, що не всі програмні середовища дозволяють звертатися до нових 64-розрядних регістрів і інструкцій. Наприклад, шоста версія Borland C++ Builder не сприймає назви 64-бітних регістрів.

Особливе функціональне призначення має *регістр прапорців* EFLAGS. Його біти (прапорці) відображають стан процесора після виконання чергової машинної команди. Більш детально прапорці регістра EFLAGS буде розглянуто у темі 4.

Особливим є також *регістр вказівника команди* EIP. Він містить адресу команди, яку слід виконати на наступному кроці роботи процесора.

Доступ до регістрів EFLAGS та EIP з боку програм користувача обмежено.

Що стосується сегментних регістрів (див. табл. 1.1), то вони і в сучасних процесорах лишились 16-розрядними, проте, починаючи з моделі i80386, крім CS, DS, SS та ES, з'явились додаткові сегментні регістри FS та GS. Слід зазначити, що на сучасних моделях процесорів сімейства x86 використання сегментних регістрів зазнало суттєвих змін. Скоріш за все, вже найближчим часом сегмент-

на модель пам'яті, а отже і життя регістрів CS, DS, ES, SS, FS і GS досягне кінця. Це пов'язано з поширенням засобів організації так званої "віртуальної пам'яті".

З кожним типом процесора пов'язана певна система *команд*, які здатний виконувати процесор цього типу. Приклади команд процесорів сімейства x86:

```
ADD   AX, 2     – додати до числа у регістрі AX десяткове число 2
                  (результат буде записано знову в AX);
CMP   AX, CX    – порівняти вміст регістрів AX та CX (результат цієї ко-
                  манди буде записано у певний біт регістра FLAGS);
JZ    Fin       – якщо відповідний біт FLAGS дорівнює нулю, перейти
                  до виконання команди з міткою Fin.
```

Зрозуміло, процесор не сприймає ці команди у формі, записаній вище. Кожна команда кодується послідовністю нулів та одиниць (так само, як усі числа та імена регістрів, що входять у запис команди). Наведені ж вище форми команд є записами так званою *мовою асемблера* для процесорів сімейства x86.

Асемблером називають мову програмування машинного рівня, терміни якої тотожні до термінів, які вживаються для опису процесора (прикладами таких термінів є конкретні імена регістрів). Отже, кожному новому типові процесора відповідає свій асемблер. Існує однозначна відповідність між командами, записаними мовою асемблера, та машинними кодами цих команд. Такої відповідності немає для інших алгоритмічних мов, що створюються як незалежні від конкретного типу комп'ютера. Тому мови асемблера називають апаратно-залежними мовами (або мовами "низького" рівня).

Якщо процесори відрізняються (зокрема, своєю розрядністю), то й мови асемблера для них теж відрізняються. Існують такі команди 32-розрядних процесорів, яких нема в 16-розрядних. Однак, прийнято, щоб усі команди процесорів "молодших" моделей могли виконуватись на "старших".

1.2. ПРИНЦИПИ АДРЕСАЦІЇ IBM PC

Кожен байт оперативної пам'яті (ОП чи RAM – Random Access Memory), до якої процесор має доступ через адресну шину, має свою унікальну фізичну адресу (його номер). Діапазон значень фізичних адрес залежить від розрядності адресної шини процесора. Для i486 та Pentium він має межі від 0 до $2^{32} - 1$ (4 Гб). Для процесорів Pentium Pro/II/III/IV цей діапазон ширший – від 0 до $2^{36} - 1$ (64 Гб).

Механізм керування пам'яттю є цілком апаратним, тобто програма не може сама сформувати фізичну адресу пам'яті на адресній шині. Їй доводиться "грати" за правилами процесора. Процесор апаратно підтримує дві моделі використання оперативної пам'яті:

– у *сегментованій моделі* для програми виділяються неперервні області пам'яті (сегменти), а сама програма може звертатися лише до даних, які містяться в цих сегментах;

– *сторінкову модель* можна розглядати як надбудову над сегментованою моделлю. В разі використання цієї моделі, оперативна пам'ять розглядається як сукупність блоків фіксованого розміру (4 Кб і більше). Основне вживання цієї моделі пов'язане з організацією віртуальної пам'яті, що дозволяє операційній системі використовувати для роботи програм простір пам'яті більший, аніж обсяг фізичної пам'яті. Для процесорів i486 та Pentium розмір можливої віртуальної пам'яті може сягати 4 Тб.

Сегментація – механізм адресації, який забезпечує існування кількох незалежних адресних просторів як в межах одної задачі, так і в системі в цілому для захисту задач від взаємного впливу. В основі механізму сегментації лежить поняття *сегмента*, який є незалежним підтримуваним на апаратному рівні блоком пам'яті. Розглядаючи сегментні регістри, ми зазначали, що для процесорів Intel, починаючи з i8086, прийнято спеціальний підхід до керування пам'яттю. Кожна програма в загальному випадку може складатися з будь-якої кількості сегментів, але безпосередній доступ вона має лише до трьох основних сегментів (коду, даних і стека), а також від одного до трьох додаткових сегментів даних. Програма ніколи не "знає", за якими фізичними адресами будуть розміщені її сегменти. Цим займається операційна система, яка сама розміщує сегменти програми в оперативній пам'яті за певними фізичними адресами і запам'ятовує значення цих адрес. Де саме запам'ятовує, залежить від режиму роботи процесора. У так званому *реальному* режимі згадані адреси записуються безпосередньо у відповідні сегментні регістри, а у *захищеному* режимі вони розміщуються у спеціальній системній дескрипторній таблиці. (Детальніше режими роботи процесора розглядаються далі).

Усередині сегмента програма оперує *відносною* адресою – а саме *зсувом* (англ. offset, рос. смещение) адресованої ділянки пам'яті відносно початку сегмента. Зрозуміло, що величина такого зсуву у байтах є числом від 0 до L, де L – розмір сегмента. Відносна адреса-зсув, який процесор використовує для доступу до даних усередині сегмента, називається *ефективним* зсувом.

Розрізняють три основні моделі сегментованої організації пам'яті:

– сегментована модель пам'яті реального (real) режиму;
– сегментована модель пам'яті захищеного (protected) режиму;
– площинна, або лінійна, модель пам'яті (flat memory model) захищеного режиму.

Розглянемо порядок формування фізичної адреси у реальному і захищеному режимах. Уточнимо термінологію. Під *фізичною адресою* розуміється адреса пам'яті, що видається на адресну шину процесора. Інша назва цієї адреси – *лінійна адреса*. Ця подвійність у назві зумовлена наявністю сторінкової моделі організації оперативної пам'яті. Дві наведені назви є синонімами лише при відключенні сторінкового перетворення адреси (у реальному режимі сторінкову адресацію завжди відключено). Сторінкова модель, як ми зазначили раніше, є надбудовою над сегментованою моделлю. У сторінковій моделі лінійна і фізична адреси мають різні значення.

Обговоримо спочатку схему формування адреси у *реальному режимі* роботи процесора. Одною з проблем, яку вирішували конструктори перших про-

цесорів i8086 та i8088, була проблема адресації оперативної пам'яті комп'ютера. 16-розрядні регістри цих процесорів могли містити максимальне ціле додатне число $2^{16} - 1$, тобто 65 535. Отже, якщо будь-яка адреса займала б точно один регістр (тобто два байти), адресний простір комп'ютера не міг би перевищувати 64 Кб. Було вирішено використати для розміщення адреси два регістри (тобто 4 байти) у наступний спосіб.

Нехай *A* є фізичною адресою (тобто номером найпершого байта) деякої ділянки оперативної пам'яті. Інформація про адресу *A* розміщується у двох 16-бітних регістрах, перший з яких містить *номер сегмента*, а другий – *величину зсуву* ділянки з адресою *A* відносно початку цього сегмента. При цьому приймається, що сегмент номер 0 починається з фізичної адреси (байта) 0, сегмент номер 1 – з фізичної адреси 16 (шістнадцяткове 10h),[4] сегмент номер 2 – з адреси 32 (20h), і так далі. Таким чином, якщо помножити номер у сегментному регістрі на число 16 (10h), здобудемо фізичну адресу сегмента.[5]

Максимальний зсув відносно початку сегмента (записаний у регістр зсуву) – це $2^{16} - 1 = 64$ Кб. Отже, *сегмент* у реальному режимі 16-розрядного процесора – це область пам'яті розміром 64 Кб з фізичною адресою, кратною числу 10h.

Для звертання до пам'яті за адресою, вказаною в парі регістрів <сегмент : зсув>, процесор обчислює фізичну адресу пам'яті за наступною схемою:

<фізична адреса> = <сегмент> × 10h + <зсув>.

Зокрема, якщо йдеться про адреси *команд* процесора, сегментна компонента розміщується у сегментний регістр CS,[6] а зсув – у регістр IP.[7] Тому ці адреси прийнято записувати через імена регістрів у вигляді **CS : IP**.

Записи адрес у формі **сегмент : зсув** вживаються також в описах пристроїв комп'ютера, лістингах програм тощо. При цьому користуються шістнадцятковими числами, наприклад: 90A5h:421Dh (відповідна фізична адреса становитиме: 90A5h × 10h + 421Eh = 94C6Eh, а в десятковому запису – 609390).

Прийнята система адресації зумовила те, що адресний простір 16-розрядних процесорів (моделей, нижчих за i386), дорівнював максимально можливій адресі:

FFFFh×10h + FFFFh = FFFFFh = 65535×16 + 65535=1114095.

Це трохи більше за 1 Мб. Для 32-розрядних процесорів, починаючи з моделі i386, розмір адресного простору зріс до 4 Гб. Подальше зростання розрядності відповідно збільшувало адресний простір, однак принципи системи адресації, описані вище, не було змінено. Таким чином, для будь-якого IBM PC-сумісного комп'ютера адреса байта оперативної пам'яті складається з двох частин: *сегмента* (*бази*) та *зсуву*.

---

[4] Нагадаємо: літера h наприкінці запису числа показує, що це число у шіснадцятковій системі числення.

[5] Звідси випливає, що можливими адресами початку сегмента є 0h, 10h, 20h ..., FFFF0h.

[6] code segment – сегмент коду

[7] instruction pointer – вказівник на команду

Недоліки такої організації пам'яті:

– сегменти безконтрольно розміщуються з будь-якої адреси, кратної 16, і, як наслідок, програма може звертатися до будь-яких адрес, у тому числі і реально не існуючих;

– сегменти мають максимальний розмір 64 Кб;

– сегменти можуть перекриватися іншими сегментами.

Бажанням ввести в архітектуру засоби для усунення вказаних недоліків зумовлена, зокрема, поява захищеного режиму, в якому працюють всі сучасні операційні системи, у тому числі Windows і Linux.

Основна ідея *захищеного режиму* – захистити виконувані процесором програми від взаємного впливу. У захищеному режимі процесор підтримує два типи захисту – за привілеями та за доступом до пам'яті. У контексті нашої теми розглянемо другий тип захисту.

Для введення будь-якого механізму захисту потрібно мати якомога більше інформації про об'єкти захисту. Для процесора такими об'єктами є виконувані ним програми. Організовуючи захист програм за доступом до пам'яті, фірма Intel не стала порушувати принцип сегментації, властивий її процесорам. Оскільки кожна програма займає один чи декілька сегментів у пам'яті, то логічно мати більше інформації про ці сегменти як про об'єкти, що реально існують у даний момент в обчислювальній системі. Якщо кожному з сегментів надати певні атрибути, то частину функцій з контролю за доступом до них можна перекласти на процесор. Це й було зроблено. Будь-який сегмент пам'яті в захищеному режимі має наступні атрибути: розташування сегмента в пам'яті; розмір сегмента; рівень привілеїв (визначає права даного сегмента відносно інших сегментів); тип доступу (визначає призначення сегмента: код, стек, дані чи системний об'єкт) та деякі інші. На відміну від реального режиму, у захищеному режимі програма вже не може безконтрольно звернутися до будь-якої фізичної адреси пам'яті. Для цього вона повинна мати певні повноваження і задовольняти певним вимогам.

Ключовим об'єктом захищеного режиму є спеціальна структура – *дескриптор сегмента*, який є 8-байтовим описувачем (англ. descriptor – описувач) неперервної області пам'яті, що зберігає перераховані вище атрибути.

Поле розміру сегмента у дескрипторі займає 20 бітів, що відповідає величині 1 Мб. Проте, для розрахунку реального розміру сегмента, крім поля розміру, використовують ще, так званий, *біт гранулярності* G дескриптора. Якщо цей G=0, то значення у полі розміру сегмента означає розмір сегмента в байтах, якщо ж G=1, – то в *сторінках*. Розмір сторінки складає 4 Кб. Отже, оскільки максимальне значення поля розміру сегмента складає $2^{20} - 1$= FFFFFh = 1 Мб, максимальний розмір сегмента у байтах – 1 Мб × 4 Кб = 4 Гб. Таким чином, розмір сегмента в захищеному режимі не є константою, і може сягати 4 Гб, тобто займати весь доступний фізичний простір пам'яті.

Виведення інформації про базову адресу сегмента і його розмір на рівень процесора дозволяє апаратно контролювати роботу програм з пам'яттю і запобігати зверненню до неіснуючих адрес або до адрес, що знаходяться поза межами дозволеного полем розміру сегмента.

Інший аспект захисту полягає в тому, що сегменти є нерівноправні в правах доступу до них. Інформація про це міститься в спеціальному байті AR, який входить до складу дескриптора. Суть цього механізму полягає в тому, що конкретний сегмент може знаходитися на одному з чотирьох рівнів привілеїв з номерами 0, 1, 2 і 3. Найбільш привілейованим є рівень 0.

Слід зазначити зміну ролі сегментних регістрів у захищеному режимі. Вони містять не адресу (як у реальному режимі), а так званий *селектор* – вказівник на відповідний елемент дескрипторної таблиці.

Отже, захищений режим має переваги над реальним режимом, оскільки він підтримує захист пам'яті, організацію віртуальної пам'яті та багато інших вдосконалень.

1.3. ПЕРЕРИВАННЯ

Система переривань будь-якого комп'ютера є його найважливішою частиною, що дозволяє швидко реагувати на події, обробку яких слід виконати негайно: сигнали від машинних таймерів, натиснення клавіш клавіатури або миші, збої пам'яті тощо.

Переривання (англ. interrupt) – сигнал, що повідомляє процесор про настання якої-небудь події. При цьому виконання поточної послідовності команд призупиняється і керування передається *оброблювачу переривання*, який реагує на подію та обслуговує її, після чого повертає управління в перерваний код. Залежно від джерела виникнення сигналу переривання поділяються на:

– *апаратні* або *зовнішні* (асинхронні) – події, які створені зовнішніми джерелами (наприклад, периферійними пристроями) та можуть відбутися в довільний момент: сигнал від таймера, мережевої карти або дискового накопичувача, натискання клавіш клавіатури, рух миші. Взаємодія процесора з усіма пристроями комп'ютера організована через апаратні переривання, відповідно до їх пріоритету (скажімо, переривання від клавіатури вважається важливішим за переривання від диска);

– *внутрішні* (синхронні) – події в самому процесорі як результат порушення якихось умов при виконанні машинного коду, наприклад: ділення на нуль, переповнення, звернення до неприпустимих адрес або неприпустимий код операції. У таких випадках процесор призупиняє виконання поточного завдання і генерує переривання з відповідним номером, яке спричиняє старт спеціальної програми операційної системи (оброблювача переривання);

– *програмні* (окремий випадок внутрішнього переривання) – ініціюються виконанням спеціальної інструкції в програмному коді. Їх можуть генерувати виконувані на комп'ютері програми (засоби такої генерації включено до мови асемблера, а також до інших алгоритмічних мов – як от Pascal чи C++). У асемблері процесорів архітектури x86 для виклику синхронного переривання є інструкція Int, аргументом якої є номер переривання. Програмні переривання, зазвичай, використовуються для звернення до функцій вбудованого програмного забезпечення (firmware), драйверів та операційної системи.

Усі джерела переривань поділено на класи, і кожному класу призначається свій рівень пріоритету запиту на переривання. Пріоритети можуть обслуговуватися як відносні і як абсолютні. Відносне обслуговування переривань означає, що, якщо під час обробки переривання надходить більш пріоритетне переривання, то це нове переривання буде оброблено тільки після завершення обробки поточного переривання. Абсолютне ж обслуговування означає, що коли під час обробки переривання надходить більш пріоритетне переривання, обробка поточного переривання призупиняється, і процесор виконує обробку щойно отриманого більш пріоритетного переривання. Після її завершення процесор повертається до виконання попередньо призупиненого переривання.

Переривання розрізняють за номерами (0, 1, …, 255). Користуються позначеннями IRQ-0, IRQ-1 тощо (IRQ – від англ. Interrupt request – "запит на переривання"). Якщо комп'ютер буде обладнано якимось новим пристроєм (наприклад, новим модемом, лазерним диском тощо), то крім фізичного приєднання цього пристрою до комп'ютера треба ще й призначити йому певне переривання. Ясно, що різним пристроям призначають різні переривання (хоча з цього правила бувають і винятки – наприклад, лазерний диск може "ділити" одне переривання IRQ-14 разом з вінчестером комп'ютера).

Хоча апаратні переривання відмінні від внутрішніх, вони включені у спільну нумерацію всіх переривань процесора. Так, апаратному перериванню IRQ-0 (яке виникає внаслідок операції ділення на 0) у цій нумерації відповідає номер 8.

У захищеному режимі обробка переривань відбувається складніше. По-перше, вводиться новий клас переривань, що генеруються самим процесором при порушеннях умов захисту, – так звані *винятки* (exceptions). Число можливих переривань, як і раніше, дорівнює 256, але 32 з них – з номерами від 00h до 1Fh – закріплено за exceptions. По-друге, замість далеких адрес у таблиці переривань використовуються дескриптори спеціальних системних об'єктів, так званих *шлюзів*. По-третє, сама таблиця переривань, яка називається IDT (Interrupt Descriptors Table), може розміщуватись за будь-якою адресою пам'яті. Переривання з векторами від 00 до 1Fh, тобто exceptions – це основа захищеного режиму. Завдяки exceptions процесор автоматично реагує на будь-які спроби порушити захист системи і дозволяє коректно відреагувати на них.

На відміну від реального режиму, у захищеному режимі x86-процесорів прикладні програми не можуть обслуговувати переривання: ця функція доступна лише операційній системі. У грамотно побудованій операційній системі жодна прикладна програма не зможе перехопити переривання, змінити захищений код (чи навіть просто прочитати його!), вийти за межу відведених для неї адрес та ін. Завдяки exceptions, операційна система може контролювати будь-які порушення умов, поставлених нею.

1.4. ПОРТИ

Найчастіше *апаратним портом* в ПК називають спеціалізований інтерфейс – роз'єм у комп'ютері, призначений для приєднання певного периферійного пристрою, наприклад: монітора, миші, принтера тощо. В сучасних комп'ютерах існують:

– COM-порт – послідовний системний порт, який зараз в основному використовується для підключення маніпулятора "миша" старих моделей або модема. У послідовному інтерфейсі біти передаються один за одним по одній лінії;

– два PS/2-порти – послідовні порти для клавіатури та миші, які по суті є аналогами COM-портів і відрізняються конструкцією роз'ємів;

– LPT-порт – паралельний системний порт, який початково призначався для підключення матричного принтера; але надалі з'явилось багато пристроїв, здатних працювати через LPT-порт: сканери, zip-приводи і т.д. В паралельному інтерфейсі паралельними лініями передається одночасно весь байт.

Унаслідок неухильного зростання обсягів передаваної інформації, пропускна спроможність перелічених вище портів стає недостатньою, і в сучасних моделях комп'ютерів вони взагалі можуть бути відсутніми;

– USB-порти (Universal Serial Bus – універсальна послідовна магістраль) – один із сучасних системних інтерфейсів для підключення різних зовнішніх пристроїв: флеш-накопичувача, миші, клавіатури, модема, принтера, сканера, мобільного телефону тощо;

– DVI-порт (для цифрового відео), і/або HDMI-порт (мультимедійний інтерфейс високої роздільної здатності для відео в форматі HD), і/або VGA-порт для підключення монітора;

– аудіо-порти для мікрофона, стереонавушників, гучномовця;

– порт локальної мережі;

– можливі також й інші порти на спеціальних пристроях (адаптерах чи контролерах), які з'єднуються з системною магістраллю материнської плати.

Портам можуть призначатись певні апаратні переривання. Наприклад, порту COM-1 призначено переривання IRQ-4, і, якщо ваша миша приєднана до COM-1, то ясно, що миша взаємодіє з процесором через переривання IRQ-4.

Якщо процесор передає (чи отримує) дані через якийсь порт, він користується *адресою* цього порту. Приміром, обмін даними з клавіатурою здійснюється за двома адресами: за одною з них розміщено черговий символ, що вводиться з клавіатури, а за іншою процесор "пересилає клавіатурі" повідомлення, що він уже прочитав черговий символ і можна пересилати (за першою адресою) наступний. Замість "адреса порту" часто також кажуть просто "порт". Отже, слід розрізняти фізичні апаратні порти від *портів-адрес*. Портів-адрес існує багато. Вони забезпечують обмін даними не лише з пристроями, приєднаними через зовнішні фізичні порти, а й з усіма пристроями комп'ютера (дисками, звуковою платою тощо).

2. КОНТРОЛЬНІ ПИТАННЯ

1. Дайте тлумачення терміну "архітектура комп'ютера". Які *різні* значення пов'язують з цим терміном?

2. Чому вивчення архітектури пов'язано з вивченням операційних систем?

3. Чим відрізняються між собою процесори різних типів?

4. Що таке CPU? Яке його призначення?

5. Від яких характеристик залежить швидкодія роботи мікропроцесора?

6. Яким є призначення регістрів і де вони фізично розташовані?

7. Що розуміють під розрядністю регістрів процесора? Які розрядності регістрів існували, а які існують у сучасних моделях процесорів?

8. Що таке "архітектурні" регістри?

9. Що спільного і чим відрізняються регістри `AH`, `AL`, `AX` та `EAX`?

10. Наведіть назви-“псевдоніми” регістрів загального призначення `EAX`, `EBX`, `ECX`, `EDX`. Чим зумовлені такі назви цих регістрів?

11. Назвіть індексні регістри. Яким є їхнє призначення?

12. Яке призначення регістра `FLAGS` ?

13. Для чого призначений стек? Охарактеризуйте особливості його організації та використання.

14. Які регістри призначені для роботи зі стеком? Охарактеризуйте призначення кожного з них.

15. Чому асемблер називають апаратно-залежною мовою програмування?

16. Чому C++ не є апаратно-залежною мовою?

17. Які моделі використання оперативної пам’яті апаратно підтримує процесор? Охарактеризуйте їх.

18. Як в процесорах x86 реалізована підтримка сегментного способу організації віртуальної пам’яті?

19. Поясніть механізм сегментації пам’яті.

20. Що розуміють під терміном “лінійна адреса”? Чим відрізняється лінійна адреса від фізичної? Чи може лінійна адреса дорівнювати фізичній?

21. Назвіть усі відомі Вам сегментні регістри. Де і яким чином використовуються ці регістри? У чому полягає відмінність їхньої роботи у реальному та захищеному режимах?

22. Яким чином формується фізична адреса пам’яті у реальному режимі роботи? Наведіть приклад.

23. Запишіть абсолютну адресу 52h *трьома* різними способами.

24. У чому полягає ідея захищеного режиму? Для чого використовується дескриптор сегмента?

25. Яким є максимальний розмір сегмента у реальному та захищеному режимах? Яким чином це впливає на максимально можливе значення адресного простору пам’яті?

26. Що називають перериванням? На які різновиди поділяються переривання залежно від джерела виникнення їхнього сигналу?

27. Роз’ясніть специфіку системи переривань у реальному режимі.

28. Що розуміють під терміном “винятки” (exceptions)?

29. У чому полягають основні принципові відмінності роботи системи переривань процесорів x86 у захищеному режимі порівняно з реальним режимом?

30. Охарактеризуйте існуючі апаратні порти у сучасних комп'ютерах.

31. Яким чином у захищеному режимі забезпечується захист адресного простору задач?

32. Що таке "рівень привілеїв" сегмента? Скільки рівнів привілеїв є у процесорі? З якою метою вони були введені?

Тема № 2

ОСНОВНІ КОМАНДИ АСЕМБЛЕРА ДЛЯ ОПРАЦЮВАННЯ ЦІЛИХ ЧИСЕЛ

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ФОРМАТ КОМАНД

Команди мови асемблер складаються з таких полів:

[*ім'я або мітка***]** *операція операнди***[;** *коментар***]**

Ім'я або мітка може складатися з латинських літер, цифр та спеціальних знаків (., ?, _, @, $), але не може розпочинатися з цифри. Якщо використовується крапка (.), то вона повинна бути першим знаком. Мітка у полі команди відокремлюється від неї двокрапкою (:). Значенням мітки є її адреса.

Операція – мнемонічний код операції асемблера.

Операнди – об'єкти, над якими чи за допомогою яких виконуються дії в командах. Машинні команди можуть чи то зовсім не мати операндів, чи мати декілька операндів. Більшість команд мають два операнди. У випадку кількох операндів вони відокремлюються один від одного комами. Операнди можна задавати у три способи:

- безпосередньо задавати в команді числовим виразом чи константою;
- регістром процесора (`AH`, `AL`, `BH`, `BL`, `CH`, `CL`, `DH`, `DL` – 8-бітові регістри загального призначення; `AX`, `BX`, `CX`, `DX` – 16-бітові регістри загального призначення; `EAX`, `EBX`, `ECX`, `EDX` – 32-бітові регістри загального призначення; `ESP`, `EBP`, `ESI`, `EDI` – базові й індексні регістри; `SS`, `DS`, `CS`, `ES` – 16-бітові сегментні регістри);
- коміркою пам'яті – це або мітка, або змінна, або вираз, який містить змінні й безпосередньо операнди.

При потребі операнд може задаватися виразом. Тобто, в асемблері існує велике розмаїття типів операндів, які можуть задаватися неявно чи міститися безпосередньо в команді, в регістрах чи в пам'яті.

В двоопераднійй машинній команді можливі такі поєднання операндів: регістр – регістр; регістр – пам'ять; пам'ять – регістр; безпосередньо операнд – регістр; безпосередньо операнд – пам'ять. *При цьому суттєвим є те, що обидва операнди водночас не можуть бути, по-перше, безпосередньо операндами, по-друге, областями пам'яті (змінними), по-третє, сегментними регістрами.*

Коментар в асемблері відокремлюється від команди крапкою з комою. Проте, в асемблері, вбудованому в інші мови, наприклад у C++ Builder, на записування коментарів поширюються правила цієї мови і програмного середовища. Важливо пам'ятати, що у асемблерних вставках у програми C++ Builder крапка з комою вже *не вважається* ознакою коментаря.

Кожна команда записується в окремому рядку і може розпочинатися не з першої позиції. Поля команди відокремлюються одне від одного пробілами чи знаками табуляції.

При написанні асемблерного коду великі й маленькі літери не відрізняються, на відміну від мови C++. Тобто, наприклад, до регістра акумулятора можна звертатися і як EAX, і як eax.

1.2. ФОРМИ ПОДАННЯ ЧИСЕЛ

В асемблері існують такі форми подання чисел:

– двійкове число – це послідовність 0 та 1, яка завершується літерою “b” чи “B”. Наприклад, 1001b (десяткове 9), 111011001b (десяткове 473);

– десяткове ціле – послідовність цифр 0 ... 9, яка м о ж е завершуватись літерою “d” чи “D”. Наприклад, однаково сприймаються 1992 та 1992d;

– шістнадцяткове число – послідовність 16-кових цифр 0, ... , 9, A, …, F, яка завершується літерою “h” чи “H”. Якщо 16-кове число розпочинається з символу A... F, то перед ним слід обов’язково записати 0 для того, щоби відрізнити число 0AH від позначення регістра AH або число 0ABCH – від ідентифікатора ABCH;

– десяткове число з рухомою крапкою записується у вигляді мантиси і порядку. Порядок відокремлюється – від мантиси літерою “e”. Наприклад, – 1.56e8 (156 000 000), 2.7e–5 (0.000 027);

– символьна константа – рядок символів поміж апострофами (') чи лапками ("). Наприклад, "Це рядок довжиною 29 символів".

1.3. ЗМІННІ

Зазвичай, змінні в асемблері оголошуються і набувають початкових значень у сегменті даних. Адреса цього сегмента зберігається в сегментному регістрі DS чи ES. У вбудованому асемблері розмірність змінної та тип, тобто кількість байтів займаної нею пам’яті, визначають за директивами DB (Define Byte – визначити байт), DW (Define Word – визначити слово (2 байти)) і DD (Define Double word – визначити подвійне слово (4 байти)). У табл. 2.1 подано директиви оголошення даних із зазначенням розмірності і типу даних.

Таблиця 2.1. **Директиви оголошення даних**

Директиви	Тип	Розмір, байт	Використовування у співпроцесорі
DB	BYTE	1	8-розрядне ціле число або символьна змінна
DW	WORD	2	16-розрядне ціле число
DD	DWORD	4	32-розрядне ціле або дійсне число

Далі наведено приклад використання директив DB, DW та DD:

```
V1 DB 121   // Змінна V1  розмірністю в 1 байт з початковим значенням 121
V2 DB 0FFH  // Змінна V2  розмірністю в 1 байт з початковим значенням 255
V3 DB 1001b // Змінна V3  розмірністю в 1 байт з початковим значенням 9
```

```
V4 DW 0AH    // Змінна V4 розмірністю в 2 байти з початковим значенням 10
V5 DD 7      // Змінна V5 розмірністю в 4 байти з початковим значенням 7
V6 DD –50    // Змінна V6 розмірністю в 4 байти з початковим значенням –50
V7 DD 0ABCDEFH // Змінна V7 розмірністю в 4 байти зі значенням 11259375
```

1.4. ОПЕРАТОРИ ДЛЯ ЗАПИСУВАННЯ ВИРАЗІВ У ОПЕРАНДАХ[8]

В асемблері визначено оператори чотирьох типів: арифметичні, відносин, логічні й атрибутивні. Вирази в операндах обчислюються під час трансляції з урахуванням пріоритетів операцій та використаних дужок. Оператори з однаковим пріоритетом обчислюються зліва направо.

Встановлено такий пріоритет операторів:

1) елементи в круглих, квадратних (непряма адресація) чи кутових дужках(<>); структурні змінні, оператори LENGTH (кількість елементів даних), SIZE (розмір даних у байтах);

2) атрибутивні оператори PTR (перевизначення типу), OFFSET (зсув адреси операнда), SEG (сегментна адреса). Наприклад:

```
MOV AX, WORD PTR [BX][SI]
```

– переслати до AX слово за адресою [BX+SI];

3) атрибутивні оператори HIGH, LOW – видають старший (молодший) байт значення виразу. Наприклад, MOV AH, HIGH A – видає старший байт змінної;

4) множення та ділення: *, / (ділення націло), MOD (остача від ділення націло, має знак діленого), SHL (зсув ліворуч, еквівалентна помноженню на 2 у відповідному степеню), SHR (зсув праворуч, є еквівалентним до ділення на 2 у відповідному степеню);

5) + (додавання), – (віднімання);

6) оператори відносин: EQ (рівність), NE (нерівність), LT (менше), GT (більше), LE (менше чи дорівнює), GE (більше чи дорівнює);

7) NOT (не) – логічний оператор, який інвертує всі біти операнда;

8) AND (логічне "І") ;

9) OR (логічне "АБО"), XOR (виняткове "АБО" чи сума за модулем 2);

10) SHORT ("короткий") – для модифіковування атрибута NEAR, який вказує, що значення мітки знаходиться в інтервалі (–128 … 127) байтів від адреси команди, яка йде за командою JMP. Це дозволяє здобути більш економний код. Формат оператора: SHORT *мітка*

1.5. КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ

З усієї сукупності цілочисельних команд процесора на лінійних ділянках програм працюють такі групи: команди пересилання даних; арифметичні команди; логічні команди; команди керування станом процесора. Спочатку розглянемо лише групу команд пересилання даних. Ці команди здійснюють пере-

[8] Не плутайте ці оператори з *командами* асемблера. Вони вживаються в *операндах* команд, і дають можливість будувати складні операнди-вирази, як показано у наступних прикладах.

силання даних з одного місця в інше, записування і зчитування інформації з портів введення-виведення, перетворення інформації, маніпулювання з адресами і вказівниками, звертання до стека.

- **MOV** *операнд1*, *операнд2*

Базова команда пересилання даних копіює вміст операнда2 в операнд1, операнд2 не змінюється. Команда `MOV EAX, ECX` є аналогічна до оператора `EAX=ECX` в C++. Як операнд2 можуть використовуватись: число (безпосередній операнд), регістр чи змінна; як операнд1: регістр чи змінна.

Приклади команд:

```
MOV X,25  // Занести безпосередній операнд 25 у пам'ять за адресою змінної X
MOV EAX,X // Копіювати значення змінної  X в регістр EAX
MOV ECX,EAX    // Копіювати вміст регістра EAX в регістр ECX
MOV Dl,X+4     // Копіювати вміст байта за адресою X+4 в регістр DL
MOV AL,BYTE PTR[EBX] // Записати в AL байт за адресою, зазначеною в EBX
```

Суттєвою особливістю команди `MOV` є те, що не можна виконувати пересилання даних безпосередньо з однієї змінної до іншої чи з одного сегментного регістра до іншого: ці операції виконують двома командами `MOV` через регістр загального призначення чи за допомогою стека командами `PUSH/POP`.

- **PUSH** *операнд*

Завантажити операнд до стека. За операнд можуть слугувати регістр, змінна чи безпосередній операнд. Приклади команд:

```
PUSHA EAX      // Занести значення регістра EAX у стек (на його вершину)
PUSHA X        // Занести значення змінної X у стек (на вершину, при цьому
               // попереднє значення проштовхується на одну позицію вниз
```

- **POP** *операнд*

Завантажити дані стека до операнда, тобто виконати дію, зворотну до команди `PUSH`. За операнд можуть слугувати регістр або змінна і, зрозуміло, не може бути число. При виконанні команди операнд ніби виштовхується зі стека догори і стек вивільняється від нього. Приклади команд:

```
POPA X         // Вивантажити значення з вершини стека у змінну X
POPA EAX       // Вивантажити 4 байти з вершини стека в регістр EAX
```

Доволі поширеним прийомом є використання пари команд `PUSH/POP` для швидкого пересилання вмісту однієї змінної до іншої. Само собою, ці змінні мають бути однакового розміру. Наприклад, для копіювання значення зі змінної `X` до змінної `Y` можна скористатись командами:

```
PUSHA X        // Спочатку занести значення змінної X у стек, а тоді
POPA  Y        // вивантажити його з вершини стека в змінну Y
```

Такий спосіб є швидшим за альтернативне пересилання через регістр з використанням двох команд `MOV`, оскільки команди `PUSH/POP` мають один операнд (а не два). Крім того, тимчасове застосування регістра `EAX` без зайвої потреби є небажаним, оскільки він може в цей час знадобитись для інших обчислень:

```
MOV EAX, X  // Занести значення X для тимчасового зберігання до регістра EAX,
MOV Y, EAX  // а вже тоді переписати його значення до  Y
```

➢ **PUSHA** чи **PUSHAD**

Команда PUSHA завантажує до стека вміст восьми регістрів у такому порядку: AX, CX, DX, BX, SP, BP, SI, DI; команда PUSHAD – EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Тобто ці команди є аналогами восьми відповідних команд PUSH.

➢ **POPA** чи **POPAD**

Завантажити дані зі стека до восьми регістрів, тобто виконати дії, зворотні командам PUSHA/POPAD. Ця та попередня команди широко використовуються при організації підпрограм. На початку підпрограми викликають команду PUSHA, а наприкінці – POPA, щоб не змінити і не втратити вміст регістрів під час виконання підпрограми.

➢ **XCHG** *операнд1*, *операнд2*

Обмін вмістом поміж операндами, який можна виконувати над *двома регістрами* чи *над регістром та змінною*. Так для обміну вмістом двох регістрів ECX та EDX можна скористатись лише однієї командою

```
XCHG ECX,EDX,
```

або аж трьома командами з використанням стека чи третього регістра для тимчасового зберігання вмісту одного з регістрів:

```
PUSH ECX                    MOV EAX,EDX
MOV ECX,EDX                 MOV EDX,ECX
POP EDX                     MOV ECX,EAX
```

➢ **LEA** *регістр*, *змінна*

Завантажити виконавчу адресу змінної до регістра.

➢ **IN** *акумулятор*, *порт*

Завантажити до регістра акумулятора (AL, AX чи EAX) вміст порту введення/виведення, номер якого або задається числом від 0 до 255, або адресується регістром DX.

➢ **OUT** *порт*, *акумулятор*

Завантажити до порту вміст акумулятора. На командах IN та OUT будується все спілкування процесора з пристроями введення/виведення – клавіатурою, вінчестером, різними контролерами тощо, і використовуються ці команди, першою чергою, у драйверах пристроїв.

1.6. АРИФМЕТИЧНІ КОМАНДИ

➢ **ADD** *операнд1*, *операнд2*

Додати до операнда1 операнд2 і записати суму до операнда1, не змінюючи операнд2. Операнд2 може бути числом, регістром чи змінною, операнд1 – регістром чи змінною, наприклад:

```
ADD EAX,X     // Додати значення X до вмісту регістра EAX; результат в EAX
ADD X,2       // Збільшити значення змінної X на 2
ADD EAX,ECX   // Збільшити значення регістра EAX на значення регістра ECX
```

- **ADC** *операнд1*, *операнд2*

Додати до операнда1 операнд2 і прапорець CF регістра прапорців процесора й записати суму до операнда1, не змінюючи операнд2. Пара команд ADD/ADC використовується для додавання чисел підвищеної точності. Приміром, для складання двох 64-бітових цілих чисел, одне з яких розташоване у парі регістрів EDX:EAX (молодше подвійне слово (біти 0 – 31) – в EAX і старше (біти 32 – 63) – в EDX), а решта – у парі регістрів EBX:ECX:

```
ADD EAX,ECX   // EAX ← EAX + ECX
ADC EDX,EBX   // EDX ← EDX + EBX + CF
```

Якщо при додаванні молодших подвійних слів відбулося перенесення зі старшого розряду (прапорець CF=1), то це буде враховано командою ADC.

- **SUB** *операнд1*, *операнд2*

Відняти від операнда1 операнд2 і записати результати до операнда1, не змінюючи операнд2. Операнд2 може бути числом, регістром чи змінною, операнд1 – регістром чи змінною:

```
SUB EAX,X     // Зменшити значення регістра EAX на значення змінної X
SUB X,2       // Зменшити значення змінної X на 2
SUB EAX,ECX   // Зменшити значення регістра EAX на значення регістра ECX
```

- **SBB** *операнд1*, *операнд2*

Відняти від операнда1 операнд2 і прапорець CF регістра прапорців процесора й записати суму до операнда1, не змінюючи операнд2. Команду можна використовувати для віднімання 64-бітових чисел в EDX:EAX та EBX:ECX аналогічно до ADD/ADC:

```
SUB EAX,ECX   // EAX ← EAX – ECX
SBB EDX,EBX   // EDX ← EDX – EBX – CF
```

- **INC** *операнд*

Збільшити операнд (регістр чи змінну) на 1 (інкремент). На відміну від команди ADD EDX,1, команда INC EDX має лише один операнд, а тому має компактніший код. Є прямим еквівалентом операції "++" в C++.

- **DEC** *операнд*

Зменшити операнд (регістр чи змінну) на 1 (декремент). Порівняно з командою SUB EAX,1, команда DEC EAX має лише один операнд, а тому її код на 1 байт менший. Є прямим еквівалентом операції "––" в C++.

- **NEG** *операнд*

Змінення знаку операнда (регістра чи змінної) на протилежний. Наведемо приклад використання команди NEG для здобуття абсолютного значення числа в регістрі EAX:

```
@M: NEG EAX   // Змінити знак на протилежний
JS @M         // Перейти на попередню команду ще раз, якщо знак від'ємний
```

- **MUL** *операнд*

Множення чисел без знаку. Помножити операнд (регістр чи змінну) на вміст акумулятора (AL, AX чи EAX, залежно від розміру операнда) й записати результат до AX, DX:AX, EDX:EAX відповідно.

- Множення чисел зі знаком має три форми, які відрізнюються кількістю операндів:

IMUL *операнд*

Помножити операнд (регістр чи змінну) на вміст акумулятора (AL, AX чи EAX, залежно від розміру операнда) й записати результат до AX, DX:AX, EDX:EAX відповідно.

IMUL *операнд1*, *операнд2*

Помножити операнд1 (регістр) на операнд2 (регістр, змінна чи число) й записати результат до операнда1.

IMUL *операнд1*, *операнд2*, *операнд3*

Помножити операнд2 (регістр чи змінна) на операнд3 (число) й записати результат до операнда1 (регістр).

Приклади команд множення:

```
IMUL X        // Збільшити значення акумулятора, помноживши його на змінну X
IMUL EAX,X    // В результаті EAX= EAX * X
IMUL EAX,X,5  // В результаті EAX= X * 5
```

- **DIV** *операнд*

Ділення цілих чисел без знаку. Поділити вміст акумулятора (AL, AX чи EAX, залежно від розміру операнда) на операнд (регістр чи змінну) й записати результат до AL, AX, EAX, а остачу – до AH, DX, EDX відповідно. Результат завжди округлюють у бік нуля, знак остачі збігається зі знаком діленого.

- **IDIV** *операнд*

Ділення цілих чисел зі знаком. Поділити вміст акумулятора (AL, AX чи EAX, залежно від розміру операнда) на операнд (регістр чи змінну) і записати результат до AL, AX, EAX, а остачу – до AH, DX, EDX відповідно.

- **CMP** *операнд1*, *операнд2*

Порівняти операнд1 з операндом2 і встановити прапорці регістра прапорців. Дія виконується шляхом віднімання операнда2 (регістра, змінної чи числа) із операнда1 (регістра чи змінної; операнди не можуть бути змінними водночас), причому результат віднімання нікуди не записується, лише встановлюються прапорці CF, OF, SF, ZF, AF та PF. Зазвичай, команду CMP використовують разом з командами умовного переходу.

Унаслідок виконання більшості арифметичних операцій здійснюється встановлення в 0 чи 1 шести прапорців регістра прапорців процесора:

– прапорець перенесення CF=1, якщо результат попередньої операції не вмістився в операнді й відбулось перенесення одиниці зі старшого біта результату (при додаванні) чи позичання одиниці до старшого розряду (при відніманні);

– прапорець допоміжного переносу AF=1, якщо відбулось перенесення одиниці з молодших 3-х розрядів результату в четвертий (при додаванні) чи позичання одиниці при відніманні;

– прапорець нуля ZF=1, якщо результат попередньої операції дорівнює 0;

– прапорець знаку SF завжди дорівнює старшому бітові результату;

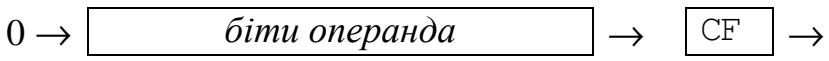
– прапорець парності PF=1, якщо найменший байт результату містить парну кількість одиничок, інакше – PF=0;

– прапорець переповнення OF=1, якщо результат попередньої арифметичної операції над числами зі знаком виходить за припустимі для них межі. Наприклад, якщо при додаванні двох додатних чисел результатом є число зі старшим бітом, рівним одиниці (тобто від'ємне число), і навпаки.

1.7. КОМАНДИ ЗСУВУ

➢ **SHR** *операнд, лічильник*

Логічний побітовий зсув операнда (регістра чи змінної) праворуч на значення лічильника (числа чи регістра CL (враховуються лише 5 молодших бітів, які набирають значень від 0 до 31). Старший біт операнда обнулюється, а молодший – заноситься до CF. Операція є еквівалентна до *беззнакового цілочисельного* ділення на 2 у степені лічильника. Ось наприклад, число 0110b (6) після зсуву на 1 праворуч дає число 011b (3), а число 1000b (8) після зсуву на 2 праворуч – число 010b (2).



Приклад використання команди:

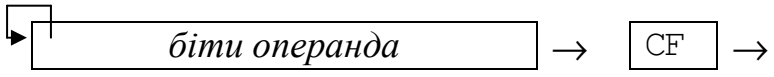
```
MOV EAX,56h    // EAX ← 86 (двійкове 0101 0110)
SHR EAX,2      // Після зсуву на 2 біти праворуч – 21 (двійкове 0001 0101), CF=1
```

Тобто, після виконання команди SHR, в регістрі десяткове значення 86 (шіснадцяткове – 56) зміниться на 21 (шіснадцяткове – 15), що є еквівалентом цілочисельного ділення цього числа на 4 (2^2): 86/4 = 21 (остача 2). Значення остачі від такого ділення втрачається, оскільки значення молодших бітів втрачаються.

Нагадаємо, що для від'ємних значень біт знаку в цій команді не зберігатиметься і братиме участь у зсуві.

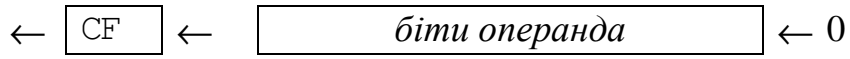
➢ **SAR** *операнд, лічильник*

Арифметичний побітовий зсув операнда (регістра чи змінної) праворуч на значення лічильника. Операція є аналогічна до SHR, але знак операнда зберігається, оскільки старший біт операнда не обнулюється, а зберігає попереднє значення.



- **SHL** *операнд*, *лічильник*
- **SAL** *операнд*, *лічильник*

Команди SHL та SAL виконують одну й ту саму дію: побітовий зсув операнда (регістра чи змінної) ліворуч на значення лічильника. Старший біт операнда заноситься до CF, а молодший – обнулюється. Операції є еквівалентні до множення на 2 у степені лічильника. Ось наприклад, число 0110b (6) після зсуву на 1 праворуч дає число 1100b (12), а число 0100b (4) після зсуву на 2 праворуч – число 0001 0000b (16).



Приклад використання команди SHL:

```
MOV AX,7  // AX ← 7 (двійкове 0111)
SHL AX,3  // Після зсуву на 3 біти праворуч – 56: (двійкове 0011 1000), CF=0
```

Тобто після виконання команди SHL, в регістрі значення 7 зміниться на 56 (шіснадцяткове – 38), що є еквівалентом множення цього числа на 8 (2^3): 7*8 = 56.

Наведемо ще один приклад застосування команд зсуву для розв'язання такої задачі. Існує *N* бактерій червоного кольору. Через 1 такт часу червона бактерія зміниться на зелену, а ще через 1 такт часу поділиться на червону та зелену. Скільки буде всіх бактерій через *k* тактів часу.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int kol,N=StrToInt(Edit1->Text),k=StrToInt(Edit2->Text);
 asm
 { mov eax,[N]
   mov ecx,[k]
   shr ecx,1
   shl eax,cl
   mov kol,eax
 } Edit3->Text=kol;
}
```

2. КОНТРОЛЬНІ ПИТАННЯ

1. Наведіть два способи збільшення вмісту регістра EAX учетверо й опишіть переваги того чи іншого способу.
2. Переведіть число 279 у шістнадцяткову та двійкову системи числення. Наведіть три команди асемблера для оголошення змінних X, Y, Z розміром 2 байти з початковим значенням 279 у цих системах числення.
3. Наведіть три-чотири способи обміну вмістом двох регістрів – EAX та ECX й опишіть переваги того чи іншого способу.
4. Що таке AH ? Наведіть приклади п'яти різних команд, які "працюють" з AH. Опишіть, які саме дії виконуватимуться у цих командах.

5. Що таке стек? Наведіть приклади чотирьох різних команд, які “працюють” зі стеком. Опишіть, які саме дії виконуватимуться у цих командах.

6. У регістрі AL міститься запис 10001001. Яке число міститься в AL? Наведіть усі можливі відповіді.

7. Чим команда MUL відрізняється від команди IMUL? Наведіть приклади різних результатів цих команд при однакових операндах.

8. У чому полягає особливість зберігання в пам’яті чисел зі знаком? Запишіть як виглядатимуть значення 255 та –3 для двобайтної знакової змінної в двійковій та шістнадцятковій системах числення.

9. Запишіть команду, яка дозволяє помножити вміст регістра EAX на 4, не вживаючи команди множення. Опишіть дію цієї команди.

10. Що спільного і чим відрізняються між собою команди ADD та ADC ? Поясніть це на прикладах.

11. Чим схоже і чим відрізяється виконання команд IMUL EAX та IMUL ECX? Куди запишуться результати?

12. Як сприйматиме асемблер записи 0AH та AH, 0CH та CH, 0BH та BH? Запишіть приклади команд з такими аргументами та опишіть, які саме дії виконуватимуться у цих командах.

13. Що таке ZF? Які команди використовують ZF? Наведіть приклади таких команд і поясніть їхню дію.

14. Запишіть команду, яка дозволяє поділити вміст регістра EAX на 8, не вживаючи операцію ділення. Опишіть дію цієї команди.

15. Запишіть десяткові значення відповідних чисел: 1001d, 1001b, 1001h. Запишіть приклади команд з такими аргументами та опишіть, які саме дії виконуватимуться у цих командах.

16. Наведіть приклади усіх відомих Вам команд, які можуть змінювати значення прапорця CF. Опишіть, які саме дії виконуватимуться у цих командах.

17. Запишіть приклади команд IN та OUT та опишіть, які саме дії виконуватимуться у цих командах.

18. Яким буде результат команди SUB EAX,EAX ? Запишіть ще декілька інших команд, які дозволяють здобути в регістрі EAX таке саме значення. Опишіть переваги того чи іншого способу.

19. Які прапорці змінять свої значення після виконання команди sub AL, AL ? Якими будуть їхні значення?

20. Що таке SF? Які команди використовують SF? Наведіть приклади таких команд і поясніть їхню дію.

21. Що таке PF? Які команди використовують PF? Наведіть приклади таких команд і поясніть їхню дію.

22. У чому подібні команди CMP та SUB ? А в чому відмінність між ними?

23. Запишіть команду обчислення модуля числа в регістрі EAX. Порівняйте і запишіть розмір в байтах цієї команди та аналогічної за дією функції abs() в C++.

24. Чим схоже і чим відрізяється виконання команд `imul EAX,ECX` та `imul ECX`? Куди запишуться результати? Порівняйте їхні розміри.

25. Поясніть дію команди `DEC EAX`. Випишіть вигляд цієї команди з вікна вбудованого дизасемблера та визначте її розмір. Порівняйте його з розміром та виглядом в дизасемблері аналогічної за дією операції декремента (`--`) в C++.

26. Продемонструйте на конкретних прикладах, чим схоже і чим відрізяється виконання команд `SHR` та `SAR`.

27. Запишіть команду інкремента числа в регістрі `EAX`. Випишіть вигляд цієї команди з вікна вбудованого дизасемблера та визначте її розмір. Порівняйте його з розміром та виглядом в дизасемблері аналогічної за дією операції інкремента (`++`) в C++.

28. Поясніть на конкретних прикладах, у чому схожі і чим відрізняється виконання команд `PUSH`, `PUSHA` та `PUSHAD`.

29. Наведіть різні способи обміну вмістом двох змінних – `X` та `Y` – й опишіть переваги того чи іншого способу.

30. Даведіть два способи збільшення вмісту регістра `EAX` на 1 та два способи зменшення вмісту регістра `EAX` на 1. Опишіть переваги того чи іншого способу.

Лабораторна робота № 1

АРИФМЕТИЧНІ ОПЕРАЦІЇ НАД ЦІЛИМИ ЧИСЛАМИ У ВБУДОВАНОМУ В C++ BUILDER АСЕМБЛЕРІ

Мета роботи: Вивчення арифметичних команд мови асемблер та набуття навичок створювання програм C++ Builder з використанням вбудованого асемблера.

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ПОЧАТКОВІ ВІДОМОСТІ ПРО ВБУДОВАНИЙ У C++ BUILDER АСЕМБЛЕР

Мова низького машинного рівня – асемблер – це мова, якою "розмовляє" процесор. Її фундаментальною перевагою перед мовами високого рівня є швидкість. Більшість програм, які працюють у режимі реального часу, або написані мовою асемблер, або використовують у критичних ділянках програмного коду асемблерні модулі.

Асемблер, вбудований у мови високого рівня, використовують для досягнення високої продуктивності роботи програм. Зручність, вигідність та легкість використовування вбудованого асемблера важко переоцінити, оскільки він не потребує окремого компілювання, компонування блоків асемблерного коду. Це є надто важливо для швидкого розробляння та налагоджування програм.

Вбудована у програму C++ послідовність асемблерних команд розміщується у фігурних дужках {} після ключового слова `asm`. Схематично блок команд на асемблері виглядає так:

```
asm
{   асемблерні команди
}
```

1.2. ПОКРОКОВЕ ВИКОНУВАННЯ ПРОГРАМИ ТА ЗАСТОСУВАННЯ ДИЗАСЕМБЛЕРА ПРИ НАЛАГОДЖЕННІ ПРОГРАМНОГО КОДУ

Застосовування точок зупину (breakpoint – переривання виконання) є потужним засобом при налагодженні програм, які дозволяють виконувати програму фрагментами від однієї точки переривання до іншої. Щоби поставити точку переривання, достатньо у вікні редактора коду клацнути мишею на смужці ліворуч коду потрібного рядка, після чого рядок набуде червоного кольору і ліворуч від нього з'явиться невеличкий червоний кружечок. При повторному клацанні мишею точка переривання зникне.

Для покрокового проходження фрагмента програми можна використовувати команди меню **Run** подані у табл. 1.1.

Таблиця 1.1. **Команди меню Run**

Команда меню Run	**“Гаряча” клавіша**	**Пояснення**
Step Over (кроками без заходу до ...)	F8	Покрокове виконання рядків програми, вважаючи виклик функції (підпрограми) за один рядок, тобто входження до функцій (підпрограм) не провадиться
Trace Into (трасування із заходом до ...)	F7	Покрокове виконання рядків програми із заходом до функцій (підпрограм), що викликаються
Trace to Next Source Line (Трасування до наступного рядка)	Shift + F7	Перехід до наступного рядка
Run to Cursor (виконати до курсора)	F4	Виконати фрагмент програми до команди, на якій розташовано курсор у вікні редактора коду

Вікно вбудованого дизасемблера відкривається за допомогою команд меню **View** (Переглянути) / **Debug Windows** (Вікна налагоджувача) / **CPU** (Процесор) при запущеному на виконання проекту (рис. 1.1).

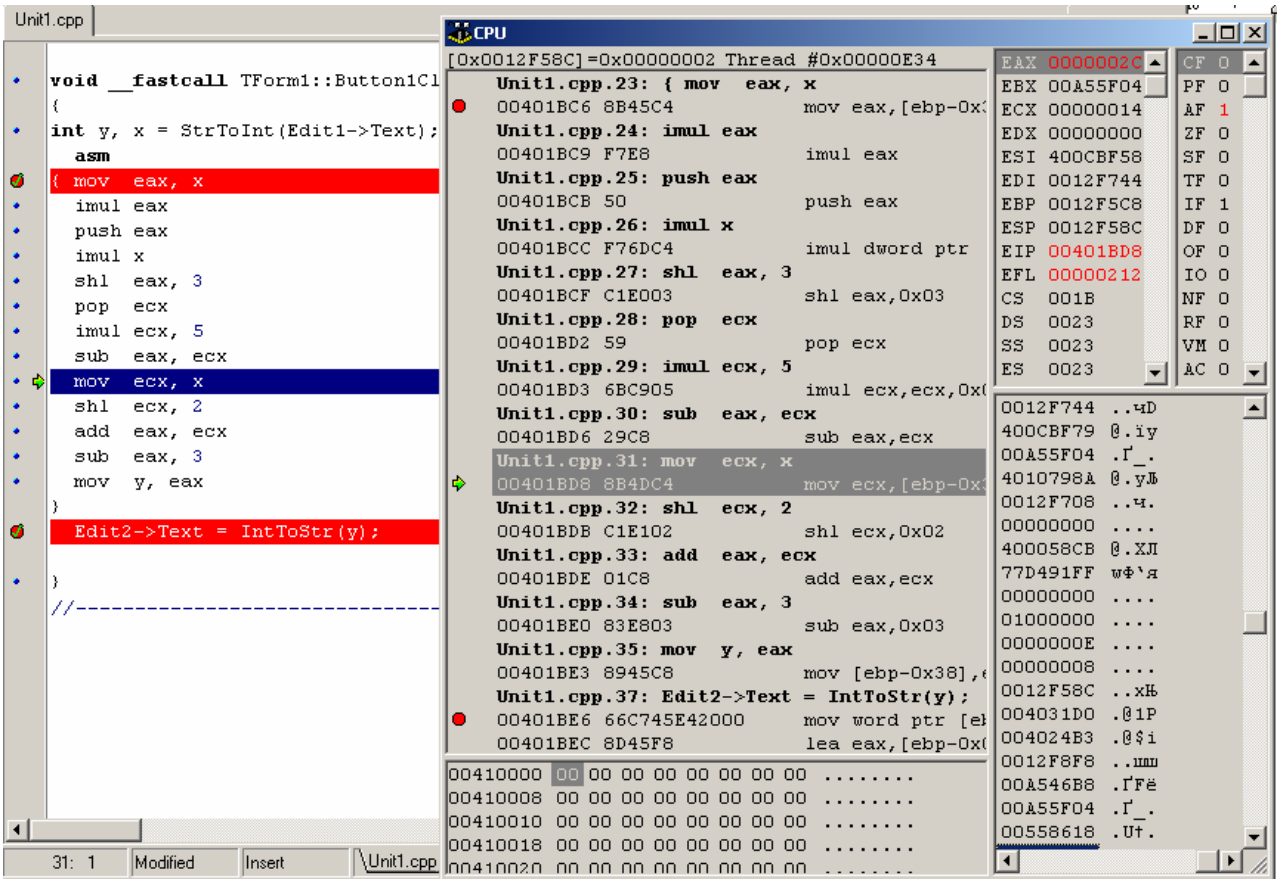


Рис. 1.1. Вигляд вікна дизасемблера для розглядуваного нижче прикладу 1.1

Для покрокового виконання програми у дизасемблері доцільно виконати таку послідовність дій:

– поставити точку переривання у першому виконуваному рядку;
– запустити покрокове виконання програми за допомогою **F8** або **F7**;
– відкрити вікно дизасемблера **View / Debug Windows / CPU**;
– здійснювати покрокове виконання програми за допомогою **F8** або **F7**, переглядаючи асемблерні коди програми та вміст усіх регістрів мікропроцесора.

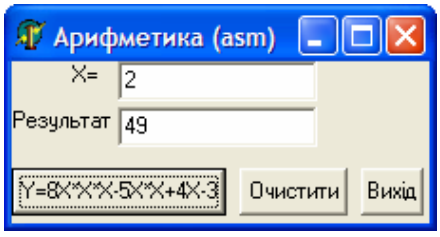
1.3. ПРИКЛАДИ ПРОГРАМ ІЗ ВБУДОВАНИМ АСЕМБЛЕРОМ ДЛЯ ВИКОНАННЯ АРИФМЕТИЧНИХ ОПЕРАЦІЙ

Приклад 1.1. Обчислити значення виразу $y = 8x^3 - 5x^2 + 4x - 3$ для різних значень x.

Вигляд вікна вбудованого дизасемблера для покрокового налагодження даного прикладу показано на рис. 1.1.

Перша команда поданого на рис. 1.1 асемблерного коду має адресу 00401BC6, а розпочинаючи з адреси 00401BE7 записується команда виведення значення до компонента Edit2. Отже, розмір усієї асемблерної вставки можна обчислити як різницю цих адрес: 00401BE7 – 00401BC6 = 21h = 33 байти.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int y, x = StrToInt(Edit1->Text);   // Оголошення цілих змінних,
  // тобто виділення пам'яті по 4 байти під кожну з них. При цьому в пам'ять
  // з адресою змінної x записується конкретне значення, введене з Edit1.
  asm
  { mov  eax, x      // Завантаження значення змінної x до регістра eax
    imul eax         // Обчислення добутку: eax*eax=x^2, результат в edx:eax,
    // при цьому в edx записується старша частина результату, якщо він перевищує
    // значення 2^31–1=2 147 483 647, інакше в edx записується 0. Надалі вважатимемо,
    // що результат не перевищує цього значення, а тому значенням edx знехтуємо.
    push eax         // Завантаження значення x^2 до стека (на його вершину)
                     // для тимчасового зберегіння
    imul eax, x      // Обчислислення добутку: eax=eax*x=x^3
    shl  eax, 3      // Логічний зсув ліворуч на 3 біти:  eax=eax*2^3=8x^3
    pop  ecx   // Вивантаження зі стека значення x^2 до ecx (стек опорожнився)
    imul ecx, 5      // Множення ecx на 5:  ecx=ecx*5=5x^2
    sub  eax, ecx    // Віднімання:  eax=eax–ecx=8x^3–5x^2
    mov  ecx, x      // Завантаження значення x до ecx
    shl  ecx, 2      // Логічний зсув ліворуч на 2 біти:  ecx=ecx*2^2=4x
    add  eax, ecx    // Додавання  eax=eax+ecx=8x^3–5x^2+4x
    sub  eax, 3      // Віднімання: eax=eax–3=8x^3–5x^2+4x–3
    mov  y, eax      // Копіювання обчисленого значення до змінної y
  }
  Edit2->Text = IntToStr(y);
}
```



За результатами покрокового виконання команд цієї програми при $x = 2$ стан використовуваних регістрів подано в табл. 1.2. Стан регістрів подано у шістнадцятковій системі, а в дужках подано їхнє десяткове значення для двознакових чисел.

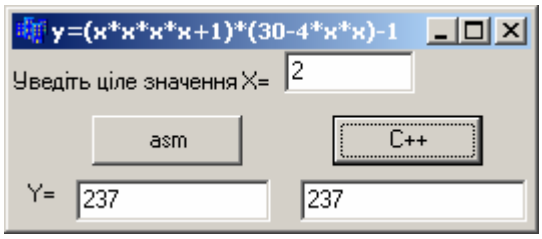
Таблиця 1.2. **Стан регістрів процесора при покроковому налагодженні**

Команди	Стан використовуваних регістрів		Стек
	EAX	ECX	
mov eax, x	**2**		
imul eax	**4**		
push eax	4		**4**
imul eax, x	**8**		4
shl eax, 3	**40h (=64)**		4
pop ecx	40h (=64)	**4**	
imul ecx, 5	40h (=64)	**14h (=20)**	
sub eax, ecx	**2Ch (=44)**	14h (=20)	
mov ecx, x	2Ch (=44)	**2**	
shl ecx, 2	2Ch (=44)	8	
add eax, ecx	**34h (=52)**	8	
sub eax, 3	**31h (=52)**	8	
mov y, eax	31h (=49)	8	

Приклад 1.2. Обчислити значення виразу $y = (x^4 + 1)(30 - 4x^2) - 1$.

Для зручності перевірки правильності обчислень в асемблері створимо на формі додаткову кнопку й розробимо програмний код для обчислення завдання мовою C++. Окрім того, порівняємо розміри займаної пам'яті програмних кодів асемблера та C++.

```
void __fastcall TForm1::Button1Click(TObject *Sender) // asm
{ int y, x = StrToInt(Edit1->Text);
  asm
  { mov   eax, x    // eax ← x
imul  eax           // eax ← x^2
push  eax           // Зберегти x^2 в стеку
imul  eax           // eax ← x^4
inc   eax           // eax ← x^4+1
pop   ecx           // ecx ← x^2
imul  ecx,4         // ecx ← 4x^2
neg   ecx           // ecx ← (–4x^2)
add   ecx,30        // ecx ← 30–4x^2
imul  ecx           // eax ← (x^4+1)( 30–4x^2)
dec   eax           // eax ← (x^4+1)( 30–4x^2)–1
mov  y, eax
  }
  Edit2->Text = IntToStr(y);   }
```



```
void __fastcall TForm1::Button2Click(TObject *Sender) //C++
{ int y, x = StrToInt(Edit1->Text);
  y = (x*x*x*x + 1)*(30 - 4*x*x) - 1;
  Edit2->Text = IntToStr(y);
}
```

У наведеному прикладі 1.2 обчислюємо розміри обох кусків програмного коду, для цього встановлюємо чотири точки зупину на початку та наприкінці кожного з розв'язань (рис. 1.2), визначаємо адреси перших та останніх команд та обчислюємо різниці цих адрес. Для асемблерної вставки код першої команди міститься за адресою 00401BC6, а останньої – 00401BDE. Отже, розмір асемблерного коду вставки становить 00401BDE – 00401BC6 = 18 h = 24 байти. Обчислення команди `y=(x*x*x*x+1)*(30-4*x*x)-1;` у C++ займає пам'ять від 00401CB6 по 00401CDD (рис. 1.2), тобто 00401CDD – 00401CB6 = 27 h = = 39 байтів, що суттєво більше розміру всієї асемблерної вставки.

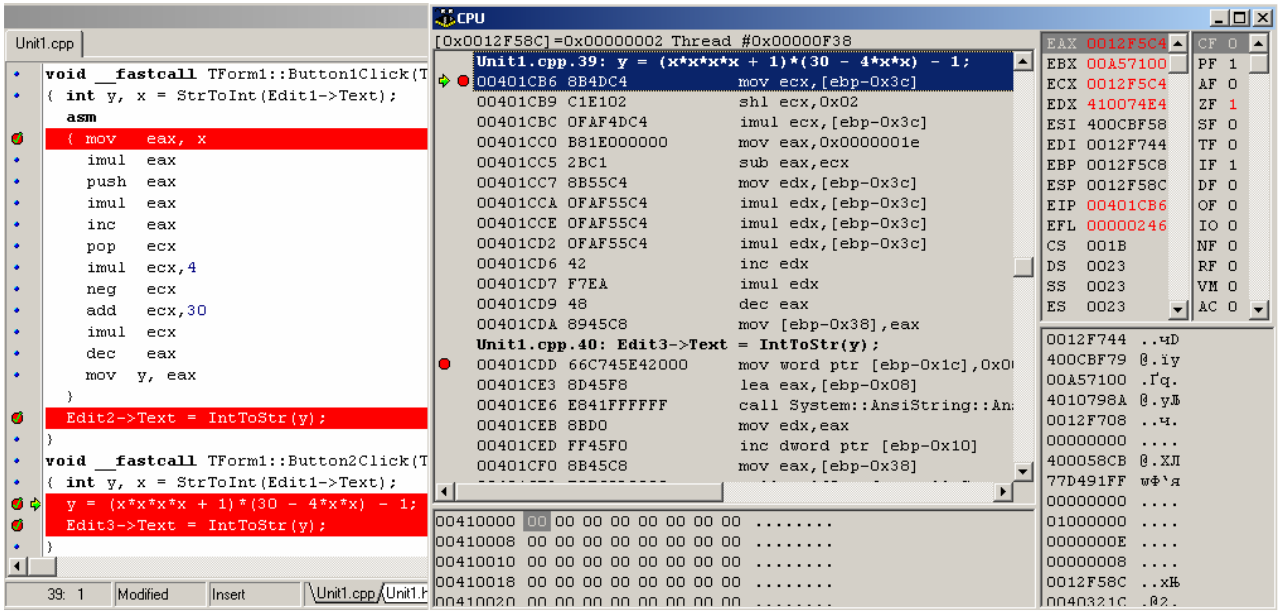


Рис. 1.2. Вигляд вікна дизасемблера для прикладу 1.2

2. ЛАБОРАТОРНЕ ЗАВДАННЯ

1. Розробити програмний проект C++ Builder для введення початкових даних і розв'язання засобами вбудованого асемблера індивідуального завдання відповідно до варіанта з табл. 1.3. Як елементи керування використовувати командні кнопки: для запускання обчислень; для очищування текстових полів форми; для завершення роботи з програмою.

2. Створити таблицю на зразок табл. 1.2 та заповнити значеннями вмісту використовуваних у проекті регістрів і стека для кожної команди програми. Для заповнення і перевірки значень таблиці у процесі відлагодження програми використовувати покрокове виконування команд й відстежувати стан стека та регістрів мікропроцесора у вікні дизасемблера.

3. Для зручності перевірки правильності обчислень в асемблері створити додаткову кнопку й написати програмний код для обчислення індивідуального завдання на C++ на зразок прикладу 1.2.

4. Обчислити та порівняти розміри програмного коду в C++ та асемблерної вставки на зразок прикладу 1.2.

5. Оформити протокол лабораторної роботи, в якому, окрім відповідей на відповідні контрольні запитання (вибираються згідно з індивідуальним варіантом на стор. 18 та на стор. 28) і тексту програми, записати таблицю (див. п. 2) та заповнити її. Занести результати обчислень до протоколу.

Таблиця 1.3. **Варіанти індивідуальних завдань**

№ вар.	Індивідуальне завдання	№ вар.	Індивідуальне завдання
1	$Y = 4X\,(X^2 + 3) - 7X^2 - 1$	16	$Y = X\,(X^2 - 1) + 11X^2$
2	$Y = 2(X^4 - X^2) + 1$	17	$Y = (X^2 - 2X^4 - 5)\,(4X + 1)$
3	$Y = 8X\,(1 - X^2) - 7$	18	$Y = 2X\,(1 - X^2) + 11(X^3 - 5)$
4	$Y = 4(X^4 - 3X^2) + 2X + 1$	19	$Y = 8(X^2 - 3)\,(8X + 1) - X^2$
5	$Y = (X^2 - 2)(2X^2 - 1) + 5$	20	$Y = (9 - 2X^2)\,(2 - X)^2 + 1$
6	$Y = (3 - X)^2\,(4X - 1) + 4X$	21	$Y = (2X^2 - 11)^2 - 2X^2 + 1$
7	$Y = (8X + 1)(X^2 - 6) - 1$	22	$Y = (X + 1)(4X^2 - 9) - (X + 1)^2$
8	$Y = 2X^3 + 3X^2 - 15X + 7$	23	$Y = 4X^3 - 6X^2 + X + 1$
9	$Y = X^3 - 11X^2 + 8X + 1$	24	$Y = X^3 - 3X^2 + 8(X + 1)^2$
10	$Y = X^3 - X^2 + 3X - 1$	25	$Y = (X^3 + 2)(8X^2 - 3) - 1$
11	$Y = 4X^3 - 8X^2 + X + 1$	26	$Y = 2X^2\,(5X^2 - 9) + 3X^3$
12	$Y = (X^2 - 1)(2 + X) + 3X^2$	27	$Y = 3X^2\,(4X^2 - 5) - 4X^2 + 15$
13	$Y = (8 - X)^2\,(4X - 1)$	28	$Y = (11X^2 - 5)(X^2 + 1) - 1$
14	$Y = (8X + 1)(X^2 - 6) - X^2$	29	$Y = (X^3 - 7)\,(9 - 2X^2) + 1$
15	$Y = (X^4 + 1)(3 - 4X^2) + 1$	30	$Y = 3X^3 - 8X^2 - X + 1$

Тема № 3

РОБОТА В АСЕМБЛЕРІ З ДІЙСНИМИ ЧИСЛАМИ

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

У процесорах Intel усі операції з дійсними числами (числами з рухомою крапкою) виконує співпроцесор (FPU – Floating Point Unit) – пристрій з власними регістрами та набором команд. Окрім роботи з дійсними числами, співпроцесор підтримує роботу з цілими числами. FPU виконує усі обчислення у 80-бітовому розширеному форматі, а 32- й 64-бітові числа використовуються для обміну даними з основним процесором та пам'яттю.

1.1. СПІВПРОЦЕСОРНІ КОНФІГУРАЦІЇ

Використання співпроцесора дозволяє значно прискорити роботу програм з обчисленнями високої точності, тригонометричні обчислення тощо. Співпроцесор (FPU) підключається до системної шини паралельно з центральним процесором (CPU) і може працювати тільки разом з ним. Усі команди попадають в обидва процесори, але кожен процесор виконує тільки свої команди. Співроцесор не має своєї "окремої" програми і не може здійснювати вибирання команд і даних. Це робить центральний процесор. FPU перехоплює з шини дані і після цього виконує відповідну команду. Два процесори працюють паралельно, що підвищує ефективність системи. Однак, виникають ситуації, коли їхня робота потребує синхронізації (через різницю у часі виконання команд).

Синхронізація за командами. Коли CPU вибирає для виконання команду FPU, останній може бути зайнятий виконанням попередньої команди. Тому перед кожною командою співроцесора в програмі має стояти спеціальна команда (wait), яка тільки перевіряє поточний стан FPU і, якщо він зайнятий, переводить CPU у стан очікування. Команду wait можна розмістити в асемблер-програмі, але якщо вона відсутня, компілятор сам вставить її у потрібних місцях.

Синхронізація за даними. Якщо виконувана в FPU команда записує операнд у пам'ять перед наступною командою CPU, яка звертається до цієї комірки пам'яті, необхідна команда перевірки стану FPU. Якщо дані ще не були записані, процесор має перейти у стан очікування. Автоматично врахувати такі ситуації доволі складно, і саме тому програміст сам повинен задавати команди, які перевіряють стан співпроцесора та за потреби змушують центральний процесор очікувати.

1.2. РЕГІСТРИ FPU

FPU має вісім 80-бітових регістрів для зберігання даних і п'ять допоміжних регістрів. Вісім регістрів даних розглядаються як стек, вершина якого називається `ST(0)`; більш глибокі елементи – `ST(1)`, `ST(2)`, … , `ST(7)`. В організації регістрового стека співроцесора є певні відмінності від класичного стека, зокрема:

– у командах співпроцесора допускається явне чи неявне звертання до регістрів. Наприклад, команда `fsqrt` замінює число у вершині стека `ST(0)` значенням квадратного кореня з `ST(0)`. В бінарних операціях допускається явне іменування регістрів, наприклад: `FSUBR ST(1),ST(2);`

– з кожним регістром стека асоціюється 2-бітовий тег (етикетка, ярлик), сукупність яких утворює слово-регістр тегів. Тег регістра `ST(0)` міститься в молодших бітах, `ST(7)` – в старших бітах слова тегів. Тег фіксує наявність у регістрі дійсного числа (код 00), нуля (код 01), ненормалізованого значення чи нескінченності (код 10), і відсутність даних (код 11). Наявність тегів дозволяє співроцесору швидше виявляти особливі випадки (спроба завантаження значення в непустий регістр, спроба вивантаження з пустого) і опрацьовувати дані.

Рештою регістрів співроцессра є регістр керування, регістр стану, два регістри стану команди і два регістри вказівника даних. Довжина їх усіх 16 біт.

1.3. ЧИСЛА FPU

Співроцесор підтримує дійсні числа з рухомою крапкою таких форматів:

Назва	**Розмір** (у бітах)	**Розмір мантиси**	**Зсвув порядку**	**Діапазон нормалізованих значень**	
single precision одинарна точність	32	23	127	2^{-126}	2^{127}
double precision подвійна точність	64	52	1 023	$2^{-1\,022}$	$2^{1\,023}$
double extended precision подвійна розширена точність	80	64	16 383	$2^{-16\,382}$	$2^{16\,383}$

При завантаженні числа співроцесор переводить його у формат подвійної розширеної точності.

Формат дійсного числа з рухомою крапкою такий: із зсувом нуль розташована мантиса, далі – експонента + зсув порядку, і старший біт – знак (0 – додатне, 1 – від'ємне). Розміри чисел зазначено у таблиці вище. 32- та 64-бітні числа не містять перший біт мантиси, він вважається за одиницю. Отже, числа 32-, 64- та 80-бітного розміру мають 24, 53 та 64 біти для зберігання мантиси числа. Наприклад, переведемо число 123.375 в формат з рухомою крапкою. Ціла частина 123 = 111 1011_2. Дробова частина обчислюється так:

$$0.375 * 2 = 0.75 \quad < 1 \quad \to 0$$
$$0.75 * 2 = 1.5 \quad \geq 1 \quad \to 1$$
$$(1.5 - 1) * 2 = 1 \quad \geq 1 \quad \to 1$$

Отже, $0.375 = 0.011_2$. Як бачите, кожного разу при множенні на 2 окремо записується біт (0 чи 1) цілого результату, допоки число не нуль або ще лишилися вільні біти числа. Якщо ж кількість вільних бітів відома, можна помножити дійсну частину на відповідну степінь двійки, не видаляючи провідні нулі.

Вийшло, що число 123.375 дорівнює 1111011.011_2. Перенесемо рухому крапку, поставивши її після першого біта числа, для цього помножимо його на такий степінь двійки, щоб вийшло число видгляду 1.xxx, – в нашому разі це степінь 6:

$$1111011.011 = 1.111011011 * 2^6$$

Число 1.111011011 називається мантисою, 6 – експонентою, 1 – ціла частина мантиси.

32 біти: `0100 0010 1111 0110 1100 0000 0000 0000`

64 біти: `0100 0000 0101 1110 1101 1000 0000 .... 0000`

80 бітів: `0100 0000 0000 0101 1111 0110 1100 0000 .... 0000`

1.4. ДЕЯКІ КОМАНДИ FPU

1.4.1. Команди пересилання даних

- **FLD** *змінна* // *Завантажити дійсну змінну до стека в ST(0)*
- **FILD** *змінна* // *Завантажити цілу змінну до стека в ST(0)*
- **FST** *змінна* // *Скопіювати ST(0) до дійсної змінної*
- **FSTP** *змінна* // *Вивантажити (виштовхнути) ST(0) до дійсної змінної*
- **FIST** *змінна* // *Скопіювати ST(0) до цілої змінної*
- **FISTP** *змінна* // *Вивантажити (виштовхнути) ST(0) до цілої змінної*
- **FXCH** *регістр* // *Змінити місцями два регістри (ST(0) і регістр)*
  // *для FXCH (без операндів) – ST(0) та ST(1)*

1.4.2. Константи FPU

- **FLD1** // *Завантажити до ST(0) число 1.0*
- **FLDZ** // *Завантажити до ST(0) число 0.0*
- **FLDPI** // *Завантажити до ST(0) число* π
- **FLDL2E** // *Завантажити до ST(0) число* $Log_2(e)$ *(≈1,4427)*
- **FLDL2T** // *Завантажити до ST(0) число* $Log_2(10)$ *(≈3,322)*
- **FLDLN2** // *Завантажити до ST(0) число Ln(2) (≈0,69315)*
- **FLDLG2** // *Завантажити до ST(0) число Lg(2) (≈0,30103)*

1.4.3. Базові арифметичні команди

- **FADD** *операнд1*, *операнд2*

Додавання до операнда1 значення операнда2 і розміщення результату в операнді1. Команда може мати такі форми:

1) `FADD X` // *ST(0) ← ST(0) + X, де X – дійсна змінна*
2) `FADD ST(0),ST(n)` // *Коли обидва операнди є регістрами,*
   `FADD ST(n),ST(0)` // *де n – номер одного з семи регістрів від 1 до 7*
3) `FADD` // *Без операндів діє аналогічно до FADD ST(0),ST(1)*

- **FIADD** *ціла змінна* // *ST(0) ← ST(0) + ціла змінна*
- **FSUB** *операнд1*, *операнд2* // *операнд1 ← операнд2 – операнд1*

Віднімання з операнда1 значення операнда2 і розміщення результату в операнді1. Команда може мати такі форми:

1) `FSUB X` // *ST(0) ← ST(0) – X, де X – дійсна змінна*
2) `FSUB ST(0),ST(n)` // *Коли обидва операнди є регістрами,*
   `FSUB ST(n),ST(0)` // *де n – номер одного з семи регістрів від 1 до 7*

Наприклад команда `FSUB ST(0),ST(1)` виконуватиметься так: `ST(0) ← ST(0) – ST(1)`.

3) FSUB

Без операндів ця команда діє аналогічно до `FSUB ST(1),ST(0)`, тобто `ST(0) ← ST(1) – ST(0)`.

- **FISUB** *ціла змінна* // *ST(0) ← ST(0) – ціла змінна*
- **FSUBR** *операнд1*, *операнд2*

Обернене віднімання із операнда2 значення операнда1, наприклад:

```
FSUBR ST(0),ST(1)   // ST(0) ← ST(1) – ST(0)
FSUBR X             // ST(0) ← X – ST(0), де X – дійсна змінна
FSUBR               // ST(0) ← ST(0) – ST(1)
```

- **FISUBR** *ціла змінна* // *Обернене віднімання: ST(0) ← ціла змінна – ST(0)*
- **FMUL** *операнд1*, *операнд2*

Множення дійсних значень операнда1 та операнда2. Результат записується до операнда1. Команда може мати такі форми:

1) `FMUL X` // *ST(0) ← ST(0) * X, де X – дійсна змінна*
2) `FMUL ST(0),ST(n)` // *Коли обидва операнди є регістрами,*
   `FMUL ST(n),ST(0)` // *де n – номер одного з семи регістрів від 1 до 7*

Наприклад команда `FMUL ST(0),ST(1)` виконуватиметься так: `ST(0) ← ST(0) * ST(1)`.

3) `FMUL` // *Без операндів діє аналогічно до FMUL ST(0),ST(1)*

- **FIMUL** *ціла змінна*

Множення цілої змінної та `ST(0)`. Результат у `ST(0)`.

- **FDIV** *операнд1*, *операнд2*

Ділення операнда1 на операнд2. Результат в операнді1. Команда може мати такі форми:

1) `FDIV X` // *ST(0) ← ST(0) / X, де X – дійсна змінна*
2) `FDIV ST(0),ST(n)` // *Коли обидва операнди є регістрами, де n – номер*
   `FDIV ST(n),ST(0)` // *одного з семи регістрів від 1 до 7*

Наприклад, команда `FDIV ST(0),ST(1)` виконуватиметься так: `ST(0) ← ST(0) / ST(1)`.

3) `FDIV` // *Без операндів діє аналогічно до FDIVP ST(1),ST(0),*
// *тобто ST(0) ← ST(1) / ST(0); ST(1) пустий*

- **FIDIV** *ціла змінна* // *ST(0) ← ST(0) / ціла змінна*
- **FDIVR** *операнд1*, *операнд2*

Обернене ділення операнда2 на значення операнда1, наприклад:

```
FDIVR ST(0),ST(1)    // ST(0) ← ST(1) / ST(0)
FDIVR X              // ST(0) ← X / ST(0), де X – дійсна змінна
FDIVR                // ST(0) ← ST(0) / ST(1)
```

- **FIDIVR** *ціла змінна* // *Обернене ділення: ST(0) ← ціла змінна / ST(0)*
- **FABS** // *Абсолютне значення ST(0): ST(0) ← |ST(0)|*
- **FCHS** // *Змінити знак значення ST(0) на протилежний*
- **FSCALE** // *ST(0) ← ST(0) * 2 $^{ST(1)}$*

Множення `ST(0)` на 2 у степені `ST(1)`. Значення `ST(1)` попередньо округлюється до меншого цілого.

- **FSQRT** // *Квадратний корінь з ST(0)*
- **FRNDINT** // *Округлити ST(0) до найближчого цілого числа*

1.4.4. Трансцендентні команди FPU

- **FSIN** // *Синус числа ST(0), заданого у радіанах*
- **FCOS** // *Косинус числа ST(0), заданого у радіанах*
- **FSINCOS** // *Синус та косинус числа ST(0), заданого у радіанах.*

Значення косинуса – в `ST(0)`, синуса – в `ST(1)`, тобто наступною командою `FDIVR` можна здобути котангенс в `ST(0)`.

- **FPTAN** // *Тангенс числа ST(0), заданого у радіанах.*

Результат – в `ST(1)`, а в `ST(0)` – 1, тобто наступною командою `FDIVR` можна здобути котангенс в `ST(0)`.

- **FPATAN** // *ST(0) ← Arctg(ST(1) / ST(0))*

Арктангенс числа, здобутого при діленні `ST(1)` на `ST(0)`. Отже, ця команда обчислює кут поміж віссю абсцис і лінією, проведеною з центра координат до точки `ST(1)`, `ST(0)`. Результат – в `ST(0)`.

- **F2XM1** // *Обчислення ST(0) ← 2 $^{ST(0)}$ – 1*

Початкове значення `ST(0)` має бути в межах від –1 до +1, інакше результат буде невизначеним.

- **FYL2X** // *Обчислення ST(0) ← ST(1) * Log2(ST(0))*

Результат у `ST(0)`, а `ST(1)` стає пустим (empty). Значення `ST(0)` має бути додатним.

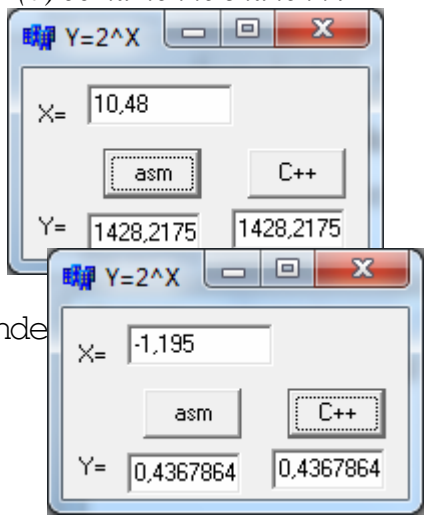
- **FYL2XP1** // *Обчислення ST(1)*Log2(ST(0)+1)*

Результат у `ST(0)`, а `ST(1)` стає пустим (empty). Початкове значення `ST(0)` має бути в межах від $-(1-\sqrt{2}/2)$ до $(1-\sqrt{2}/2)$, інакше результат буде невизначеним. Якщо аргумент логарифма близький до нуля, `FYL2XP1` дає більшу точність порівняно з `FYL2X`.

Застосуємо деякі з наведених команд для обчислення $Y = 2^X$. Якби X було цілим числом, для цього підійшла б команда `FSCALE`; а для дійсного $X \in (-1,1)$ можна було б застосувати команду `F2XM1`. Розглянемо загальний випадок довільного показника степеня X. Позначимо через *A цілу частину* числа X, а через B – його *дробову частину*. Зрозуміло, що $X=A+B$, і що $B \in (-1,1)$. Тепер скористаємось тим, для будь-яких значень A та B має місце $2^{A+B} = 2^A * 2^B$. Величину 2^A можна обчислити командою `FSCALE`; а величину 2^B-1 командою `F2XM1`.

Ці міркування обґрунтовують наступну програму обчислення $Y = 2^X$.

```
void __fastcall TForm1::Button1Click(TObject *Sender)  // pow(2,x)
{ double Y,  X=StrToFloat(Edit1->Text);
  asm
  { FLD X            // ST(0) ← X
    FLD ST(0)        // ST(1) ← ST(0) ← X, тобто в ST(0) та ST(1) дві копії значення X
    FRNDINT          // Округлити до найближчого цілого значення в ST(0)
    FXCH             // Поміняти місцями вміст регістрів ST(0) та ST(1)
    FSUB ST(0),ST(1) //B ST(0) – дробова частина числа X, в ST(1) – ціла частина X
    F2XM1            //  ST(0) ← 2^ST(0) – 1
    FLD1             // ST(0) ← 1, ST(1) ← 2^дробова частина X – 1, ST(2) ← ціла частина X
    FADD             // ST(0) ← 2^дробова частина X, ST(1) ← ціла частина X
    FSCALE           // ST(0) ← ST(0) * 2^ST(1), тобто в ST(0) остаточне значення 2^X
    FSTP Y           // Y ← ST(0)
    FINIT            // Очищення всіх регістрів FPU
  }
  Edit2->Text=FloatToStr(Y);
}
//---------------------------------------
#include <math.h>
void__fastcall TForm1::Button2Click(TObject*Sende
{ double Y,X=StrToFloat(Edit1->Text);
  Y=pow(2,X);
  Edit3->Text=FloatToStr(Y);     }
```

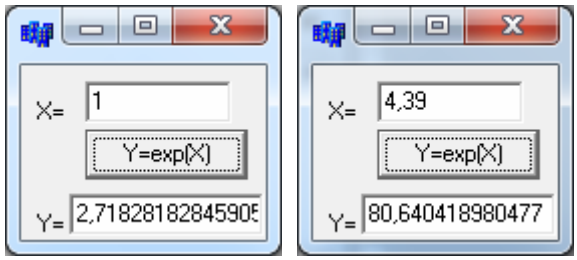


У представленій програмі спочатку обчислюються ціла та дробова частини числа X. Для цього, за допомого команди `FRNDINT` округлюємо до найближчого цілого значення в ST(0) та віднімаємо його з попередньо продубльованого значення X. Після цього команда `F2XM1` підносить число 2 до дробової частини степеня X, і віднімає від результату одиницю. Наступними двома командами компенсуємо це віднімання. Тепер у `ST(0)` міститься значення $2^{\text{дробова\_частина\_}X}$, а у `ST(1)` – ціла частина X. Скориставшись командою `FSCALE`, здобуваємо у `ST(0)` остаточне значення 2^X, і зберігаємо його в змінній Y.

Для перевірки та порівняння здобутих результатів обчислюємо аналогічний вираз у C++ за допомогою функції `pow(2,X)` (кнопка **Button2**).

Тепер, визначивши яким чином обчислити 2 у степені *X*, виконаємо обчислення e^X – експоненти від *X*, тобто числа e (≈2.7182818) у степені *X*. Скористаємось тотожністю $e^X = 2^{X*\log2(e)}$. Отже, нам треба обчислити величину $X * \log_2(e)$. Розв'язок дуже подібний до попередньої програми обчислення значення 2^X. Після завантаження в стек співпроцесора значення *X* та числа $\log_2(e)$ (≈ 1,4427) і обчислення їхнього добутку (командою `FMUL`) треба просто повторити всі дії наведеної вище програми.

```
void __fastcall TForm1::Button1Click(TObject *Sender)  // Експонента e^X
{ double Y, X=StrToFloat(Edit1->Text);
  asm
  { FLD X          // ST(0) ← X
    FLDL2E         // Завантажити до ST(0) число Log2(e) (≈1,4427)
    FMUL           // ST(0) ← X * log2 (e)
    FLD ST(0)      // ST(1) ← ST(0), тобто в ST(0) і ST(1) однакові значення
    FRNDINT        // Округлити до найближчого цілого значення в ST(0)
    FXCH           // Поміняти місцями вміст регістрів ST(0) та ST(1)
    FSUB ST(0),ST(1) // В ST(0) – дробова частина X*log2 (e), в ST(1) – ціла частина
    F2XM1          //  ST(0) ← 2^ST(0) – 1
    FLD1           // ST(0) ← 1, ST(1) ← 2 дробова частина – 1, ST(2) ← ціла частина
    FADD           // ST(0) ← 2 дробова частина, ST(1) ← ціла частина
    FSCALE         // ST(0) ← ST(0) * 2 ST(1), тобто в ST(0) остаточне значення e^X
    FSTP Y         // Y ← ST(0)
    FINIT
  } Edit2->Text=FloatToStr(Y);
}
```



При порівнянні розміру цього програмного коду та аналогічного за дією коду функції `exp(x)` у C++, стає зрозуміло, що асемблерний код є значно (≈ в 30 разів) менший.

Для обчислення значення $Y = 10^X$ слід вжити команду `FLDL2T` замість `FLDL2E` у другому рядку наведеного вище асемблерного коду (скористуємось тотожністю $10^X = 2^{X*\log2(10)}$).

Наостанок наведемо програму піднесення довільного **додатного** числа *X* до довільного степеня *Y*. Використаємо тотожність $X^Y = 2^{Y * \log2(X)}$. Величину $Y * \log_2(X)$ для будь-якого **додатного** *X* і довільного *Y* можна обчислити спеціальною командою співроцесора `FYL2X`. Решта команд буде подібною до використаних у наведених вище прикладах обчислення 2^X та e^X.

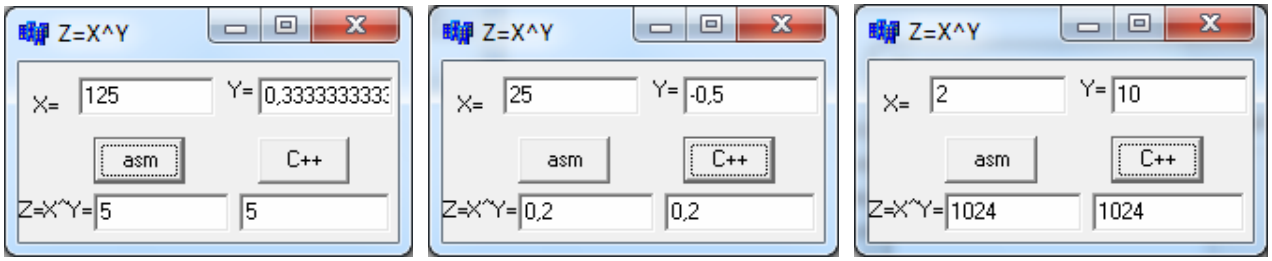
Використовуватимемо тотожності: $x^y = 2^{y * \log2(x)}$

a = (ціла частина a) + (дійсна частина a)

$a^{b+c} = a^b * a^c$

```
void __fastcall TForm1::Button1Click(TObject*Sender) // Z=X^Y для X>0
{ double Z,X=StrToFloat(Edit1->Text),Y=StrToFloat(Edit2->Text);
  asm
```

```
  { FLD Y          // ST(0) ← Y
    FLD X          // ST(0) ← X,  ST(1) ← Y
    FYL2X          // ST(0) ← ST(1) * Log2(ST(0)), ST(1) – порожній (Empty),
                   // ST(0) = y*log2(x),  ST(1) = Empty
    FLD ST(0)      // ST(1) ← ST(0), тобто у ST(0) і ST(1) те саме значення y*log2(x)
    FRNDINT        // Округлити до найближчого цілого значення в ST(0),
                   // тобто ST(0) ← int[y*log2(x)], ST(1) = y*log2(x)
    FXCH           // ST(0) ↔ ST(1)
    FSUB ST(0),ST(1) // В ST(0) – дробова частина величини y*log2(x) (позначимо
                   // її через fr(y*log2(x))), а в ST(1) – ціла частина – int[y*log2(x)]
    F2XM1          //  ST(0) ← 2^ST(0) – 1, отже ST(0)=2^fr(y*log2(x)) – 1, ST(1)=int[y*log2(x)]
    FLD1           // ST(0) ← 1, ST(1) ← 2^дробова частина – 1, ST(2) ← ціла частина
    FADD           // ST(0) ← 2^дробова частина, ST(1) ← ціла частина
    FSCALE         // ST(0) ← ST(0) * 2^ST(1), тобто в ST(0)=2^fr(y*log2(x)) * 2^int[y*log2(x)] =
     // = 2^[y*log2(x)] = x^y. Отже в ST(0) остаточне значення x^y,  ST(1)=int[y*log2(x)]
    FSTP Z         // Z ← ST(0), ST(0) ← ST(1)
    FINIT          // Очищення всіх регістрів FPU, оскільки ST(0)=int[y*log2(x)]
  }  Edit3->Text=FloatToStr(Z);
}
//---------------------------------------------------------------------
#include <math.h>
void __fastcall TForm1::Button2Click(TObject *Sender)
{ double Z,X=StrToFloat(Edit1->Text),Y=StrToFloat(Edit2->Text);
  Z=pow(X,Y);
  Edit4->Text=FloatToStr(Z);
}
```



1.4.5. Деякі команди порівняння даних FPU та обміну даними між FPU і CPU

➢ **FCOM** *операнд* // *Порівняти дійсні числа*

Порівнює `ST(0)` з операндом (дійсною змінною, FPU-регістром, або – якщо операнд не зазначено – з регістром `ST(1)`) і встановлює умовні прапорці `C0`, `C2` та `C3` регістра `SR` (слово стану співпроцесора) згідно з таблицею:

Умова	C3	C2	C0
`ST(0)` > *операнд*	0	0	0
`ST(0)` < *операнд*	0	0	1
`ST(0)` = *операнд*	1	0	0
Неможливо порівняти	1	1	1

- **FCOMP** *операнд* // *Порівняти ST(0) з операндом і виштовхнути з ST(0)*
- **FICOM** *ціла змінна* // *Порівняти ST(0) з цілою змінною, дії аналогічні до FCOM*
- **FICOMP** *ціла змінна* // *Порівняти ST(0) з цілою змінною і виштовхнути з ST(0)*
- **FTST** // *Порівняти ST(0) з нулем*

Команда `FTST` перевіряє, чи містить `ST(0)` нуль, та встановлює прапорці `C0`, `C2` та `C3` аналогічно до `FCOM`

- **FSTSW** *операнд* // *Зберегти регістр SR в операнді*

Команда зберігає поточний стан регістра `SR` (слово стану) співпроцесора в операнді (16-бітовій змінній чи регістрі `AX` процесора). Зазвичай, команда `FSTSW AX` використовується відразу після команди порівняння для виконання умовного переходу.

- **SAHF**

Вміст регістра `AH` зберігається у молодшому байті регістра `FLAGS` – прапорцях `SF` (біт 7), `ZF` (біт 6), `AF` (біт 4), `PF` (2 біт) та `CF` (біт 0) (зверніть увагу – це команда CPU).

1.4.6. Деякі команди очищення регістрів FPU

- **FFREE** `ST(n)` // *Звільнити регістр ST(n)*

Зазначений регістр `ST(n)` стає пустим (Empty), де `n` – число від 0 до 7

- **FINIT** // *Очищення всіх регістрів і всіх прапорців FPU*

Усі регістри FPU очищаються.

2. КОНТРОЛЬНІ ПИТАННЯ

1. Що таке FPU? Яким є його призначення? Чим він відрізняється від CPU? Назвіть щонайменше три суттєві відмінності між ними.
2. Чи може комп'ютер не мати співпроцесора? Іншими словами, чи є задачі, які можна розв'язати тільки з співпроцесором? Відповідь обґрунтуйте.
3. Чи може комп'ютер мати тільки співпроцесор (і не мати процесора)? Іншими словами, чи всі задачі можна розв'язати, звертаючись лише до співпроцесора? Відповідь обґрунтуйте.
4. Що таке стек співпроцесора? Чим він відрізняється (якщо відрізняється) від стека програми?
5. Скільки регістрів для зберігання даних має співпроцесор та якою є їхня ємність?
6. Наведіть два-три способи збільшення вмісту регістру `ST(0)` учетверо.
7. Скільки і які результати повертає команда `FSINCOS`? Наведіть приклади застосування цієї команди для обчислення значень тангенса та котангенса.
8. Скільки та які результати повертає команда `FPTAN`? Наведіть приклади застосування цієї команди для обчислення значень тангенса і котангенса.

9. Чому команда FPATAN має два аргументи? Наведіть приклад застосування цієї команди.

10. Як працює команда FSCALE ? Наведіть два різних приклади роботи цієї команди.

11. Наведіть два способи обчислення котангенса числа у регістрі ST(0).

12. Чим відрізняються між собою команди MUL та FMUL? Поясніть це на прикладах.

13. Чим відрізняються між собою команди FMUL та FIMUL? Поясніть це на прикладах.

14. Чим відрізняються між собою команди FSUB та FSUBR? Поясніть це на прикладах.

15. Чим відрізняються між собою команди FST та FSTP? Поясніть це на прикладах.

16. Поясніть, чому поряд з командою FST існує команда FSTP, проте поряд з командою FLD нема команди FLDP?

17. A та B є дійсними змінними. Чи допустима команда FSUB A,B ? Якщо так, який її результат? А якщо ні, поясніть, чому.

18. A та B є цілими змінними. Чи допустима команда FISUB A,B ? Якщо допустима, який її результат? А якщо недопустима, поясніть, чому.

19. X є дійсною змінною. Чи допустима команда FSUB X ? Якщо допустима, який її результат? А якщо недопустима, поясніть, чому.

20. Чим відрізняються між собою команди FDIVR та FIDIVR? Поясніть це на прикладах.

21. Як обчислити число 2^{12} за допомогою команд співпроцесора?

22. Яка команда дозволяє здобути число Ln(2)? Охарактеризуйте інші подібні команди.

23. Які дії виконує команда FLD ST(0)? Намалюйте відповідні цим діям конфігурації стека. В яких ситуаціях ця команда може спричинити помилку?

24. Які дії виконує команда FLD ST(1)? Намалюйте відповідні цим діям конфігурації стека. В яких ситуаціях ця команда може спричинити помилку?

25. Як обчислити число e (основу натуральних логарифмів) за допомогою команд співпроцесора?

26. Які дії виконує команда FLDL2E? Наведіть приклад конкретного прикладного застосування цієї команди.

27. Чим відрізняються між собою команди FSUBR та FISUBR? Поясніть це на прикладах.

28. Переведіть число 195.125 у двійковий формат та покажіть як воно розміститься у регістрі співпроцесора.

29. Які формати дійсних чисел з рухомою крапкою підтримує FPU? Чим відрізнятиметься одне й те саме дійсне число в різних форматах?

30. Які команди завантаження константних значень у стек FPU існують? Наведіть конкретні приклади використання цих команд.

Лабораторна робота № 2

АРИФМЕТИЧНІ ОПЕРАЦІЇ НАД ДІЙСНИМИ ЧИСЛАМИ У ВБУДОВАНОМУ В C++ BUILDER АСЕМБЛЕРІ

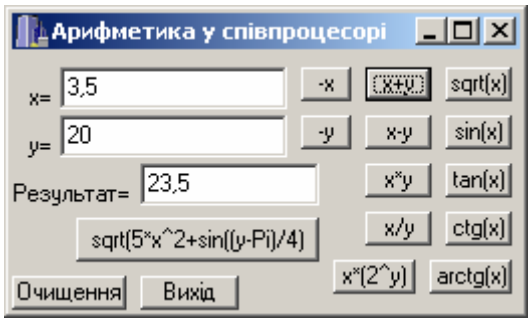
Мета роботи: Вивчення асемблерних арифметичних команд опрацювання дійсних чисел та набуття навичок роботи з регістрами співпроцесора при створенні асемблерних вставок у програмах C++ Builder.

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ПРИКЛАДИ ВИКОРИСТОВУВАННЯ КОМАНД FPU У ВБУДОВАНОМУ АСЕМБЛЕРІ

Приклад 2.1. Створити програмний проект обчислення різних математичних функцій засобами вбудованого асемблера.

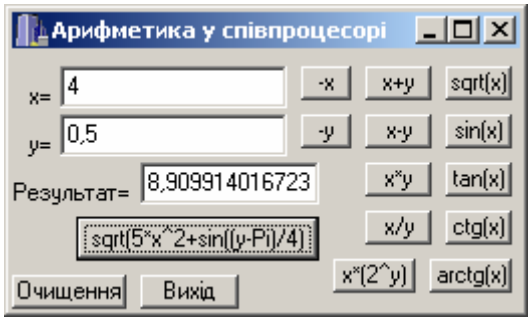
```
float x, y, z;
void __fastcall TForm1::Button1Click(TObject *Sender) // –x
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x      // ST(0) ← x
    fchs       // Змінити знак значення ST(0) на протилежний, тобто ST(0) ← (– x)
    fstp x     // x ← ST(0)
  }
  Edit3->Text = FloatToStr(x);
}
void __fastcall TForm1::Button2Click(TObject *Sender) // –y
{ y = StrToFloat(Edit2->Text);
  asm
  { fld y      // ST(0) ← y
    fchs       // ST(0) ← (– y)
    fstp y     // y ← ST(0)
  }
  Edit3->Text = FloatToStr(y);
}
```




```
void __fastcall TForm1::Button3Click(TObject *Sender) // x + y
{ x = StrToFloat(Edit1->Text);  y = StrToFloat(Edit2->Text);
 asm
 { fld x      // ST(0) ← x
   fadd y     // ST(0) ← ST(0) + y, тобто ST(0) ← x + y
   fstp z     // z ← ST(0)
 } Edit3->Text = FloatToStr(z);
}
```

```
void __fastcall TForm1::Button4Click(TObject *Sender) // x – y
{ x = StrToFloat(Edit1->Text);  y = StrToFloat(Edit2->Text);
  asm
  { fld x     // ST(0) ← x
    fsub y    // ST(0) ← ST(0) – y, тобто  ST(0) ← x – y
    fstp z    // z ← ST(0)
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button5Click(TObject *Sender) //  x * y
{ x = StrToFloat(Edit1->Text);y = StrToFloat(Edit2->Text);
  asm
  { fld x     // ST(0) ← x
    fmul y    // ST(0) ← ST(0) * y, тобто  ST(0) ← x * y
    fstp z    // z ← ST(0)
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button6Click(TObject *Sender) //  x / y
{ x = StrToFloat(Edit1->Text);   y = StrToFloat(Edit2->Text);
  asm
  { fld x     // ST(0) ← x
    fdiv y    // ST(0) ← ST(0) / y, тобто  ST(0) ← x / y
    fstp z    // z ← ST(0)
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button7Click (TObject *Sender) // sqrt(x)
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x     // ST(0) ← x
    fsqrt     // ST(0) ← √x
    fstp z    // z ← ST(0)
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button8Click(TObject *Sender) // sin(x)
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x     // ST(0) ← x
    fsin      // ST(0) ← sin(x)
    fstp z    // z ← ST(0)
  }
  Edit3->Text = FloatToStr(z);
}
```

```
void __fastcall TForm1::Button9Click(TObject *Sender) // tan(x)
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x  // ST(0) ← x
    fptan  // tan(x) в ST(1), а в ST(0) – 1
    fstp z // z ← ST(0), тобто z ← 1, при цьому tan(x) з ST(1) вишитовхується в ST(0)
    fstp z // z ← ST(0), тобто z ← tan(x)
  }        // Команди fptan та fstp z можна замінити командами fsincos та fdivr
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button10Click(TObject *Sender) // ctg(x)
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x  // ST(0) ← x
    fptan  // tan(x) в ST(1), а в ST(0) – 1
    fdivr  // ST(0) ← ST(0)/ST(1), тобто ST(0) ← ctg(x)
    fstp z
  }
  Edit3->Text=FloatToStr(z);
}
void __fastcall TForm1::Button11Click (TObject *Sender) // arctg(x)
{ x = StrToFloat(Edit1->Text);
  asm
  { fld x  // ST(0) ← x
    fld1   // ST(0) ← 1, при цьому ST(0)→ ST(1)
    fpatan // ST(0) ← arctan(st(1)/st(0))
    fstp z
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button12Click(TObject *Sender) // x * (2 ^ y)
{ x = StrToFloat(Edit1->Text);
  y = StrToFloat(Edit2->Text);
  asm
  { fld y  // ST(0) ← y
    fld x  // ST(0) ← x, при цьому y → ST(1)
    fscale // ST(0) ← ST(0) * 2^ST(1), при цьому ST(1) округляється до меншого цілого
    fstp z
  }
  Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button13Click(TObject *Sender)
                                          // sqrt(5*x^2+sin((y-Pi)/4)
{ int s;
  x = StrToFloat(Edit1->Text);   y = StrToFloat(Edit2->Text);
```

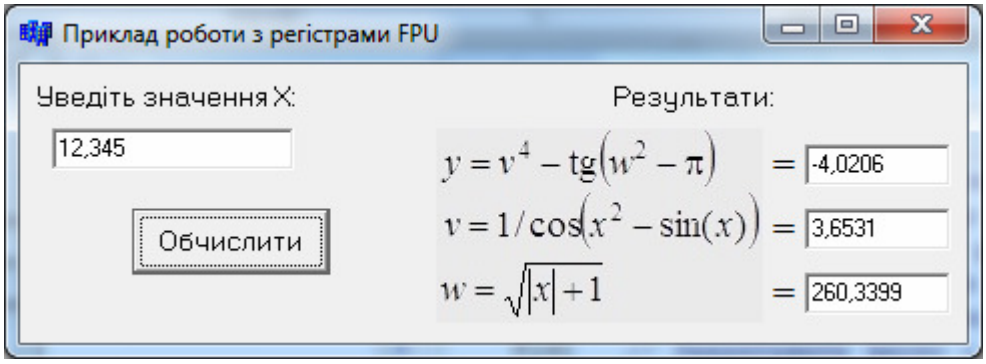



```
  asm
 { fld x
   fmul x   // ST(0) ← x * x
   mov s, 5
   fild s
   fmul     // ST(0) ← 5* x ^ 2
   fld y
   fldpi
   fsub     // ST(0) ← y – pi, a ST(1) ← 5* x ^ 2
   mov s, 4
   fild s
   fdiv     // ST(0) ← (y – pi)/4
   fsin
   fadd     // ST(0) ← 5*x^2+sin((y-Pi)/4
   fabs
   fsqrt
   fstp z   // z=sqrt(|5*x^2+sin((y-Pi)/4|)
 } Edit3->Text = FloatToStr(z);
}
void __fastcall TForm1::Button14Click(TObject *Sender) // Очищення
{  Edit1->Clear();  Edit2->Clear();  Edit3->Clear(); }
void __fastcall TForm1::Button15Click(TObject *Sender) // Вихід
{  Close();
}
```

Приклад 2.2. Створення проекту для обчислювання засобами вбудованого асемблера трьох функцій: $y = v^4 - \mathrm{tg}\left(w^2 - \pi\right)$; $v = 1/\cos\left(x^2 - \sin(x)\right)$; $w = \sqrt{|x|+1}$ для різних значень змінної *x* дійсного типу, яке вводять з вікна `Edit1`.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float x, y, v, w;
  x=StrToFloat(Edit1->Text);
  asm
  { fld x // Завантажити на вершину стека співпроцесора в регістр ST(0) float-вели-
          // чину, розміщену за адресою змінної x (float-величина у C++ займає 4 байти).
    fabs  // В ST(0) – модуль його попереднього значення.
    fld1  // Завантажити число 1 в регістр ST(0), при цьому
          // попередній вміст ST(0), тобто |x|, "опускається" в ST(1).
    fadd  // Додати до вмісту регістра ST(0) значення ST(1), результат – (1+|x|) –
          // буде в ST(0), а ST(1) стане пустим (stack is popped).
    fsqrt // В ST(0) – квадратний корінь з його попереднього значення, тобто √(|x|+1)
    fst w // Зберегти вміст регістра ST(0) за адресою змінної w.
    fld x // Завантажити float-величину, розміщену за адресою змінної x в регістр
          // ST(0), при цьому попередній вміст ST(0), тобто w, "опускається" в ST(1).
```

```
    fmul x  // Помножити значення в ST(0) на x, результат – x^2 – буде в ST(0).
    fld x   // Завантажити змінну x в регістр ST(0), при цьому попередній вміст ST(0),
            // тобто x^2, "опускається" в ST(1), а ST(1), тобто w, – відповідно в ST(2).
    fsin    // В ST(0) – синус від його попереднього значення, тобто sin(x).
    fsub    // Відняти від вмісту регістра ST(1) значення ST(0), результат – (x^2– sin(x))
            // – буде в ST(0), а в ST(1) "підніметься" значення w з ST(2).
    fcos    // В ST(0) – косинус від його попереднього значення, тобто cos(x^2– sin(x)).
    fld1    // Завантажити число 1 в регістр ST(0), при цьому
            // попередній вміст ST(0), тобто cos(x^2– sin(x)), "опускається" в ST(1).
    fdivr   // Поділити вміст регістра ST(0) на вміст регістра ST(1), результат –
            // 1/ cos(x^2– sin(x)) – буде у ST(0), а ST(1) "підніметься" значення w з ST(2).
    fst v   // Зберегти вміст регістра ST(0) за адресою змінної v.
    fmul v  // Помножити вміст регістра ST(0) на float-величину, розміщену за адресою
            // змінної v, результат – v^2 – буде в ST(0).
    fmul st(0), st(0) // Помножити ST(0) на ST(0), результат – v^4 – в ST(0).
    fxch    // Змінити місцями два регістри ST(0) та ST(1), після цього
            // в  ST(0) – w, а в ST(1) – v^4.
    fmul st(0), st(0) // Помножити ST(0) на ST(0), результат – w^2 – в ST(0).
    fldpi   // Завантажити число π в регістр ST(0), при цьому попередній вміст ST(0),
            // тобто w^2, "опускається" в ST(1), а v^4 – з ST(1) до ST(2).
    fsub    // Відняти від ST(1) значення з ST(0), результат –  w^2 – π – буде
            // в ST(0), а значення з ST(2) "підніметься" в ST(1).
    fptan   // В ST(1) – тангенс від ST(0), тобто tg(w^2 – π), а в ST(0) заноситься число 1.
    fmul    // Помножити регістр ST(0) на ST(1), результат – tg(w^2 – π) – буде в ST(0).
    fsub    // Відняти від ST(1) значення з ST(0), результат – v^4 – tg(w^2 – π) –
            // буде в ST(0), а ST(1) стане пустим (stack is popped)
    fstp y  // Виштовхнути вміст регістра ST(0) за адресою змінної y,
            // а ST(0) стає пустим (stack is popped)
      }
     Edit2->Text=FormatFloat("0.0000", v);
     Edit3->Text=FormatFloat("0.0000", w);
     Edit4->Text=FormatFloat("0.0000", y);
}
```



Приклад 2.3. Створення програми для обчислювання засобами вбудованого асемблера функцій $y = \operatorname{arctg}^3\left(x^2 - \pi\right)$; $x = 1 + \sqrt{|3 - z|}$; $z = \operatorname{ctg}^2\left(a/b^3\right)$ для різних значень дійсних змінних a та b, які вводять з вікон Edit1 та Edit2.

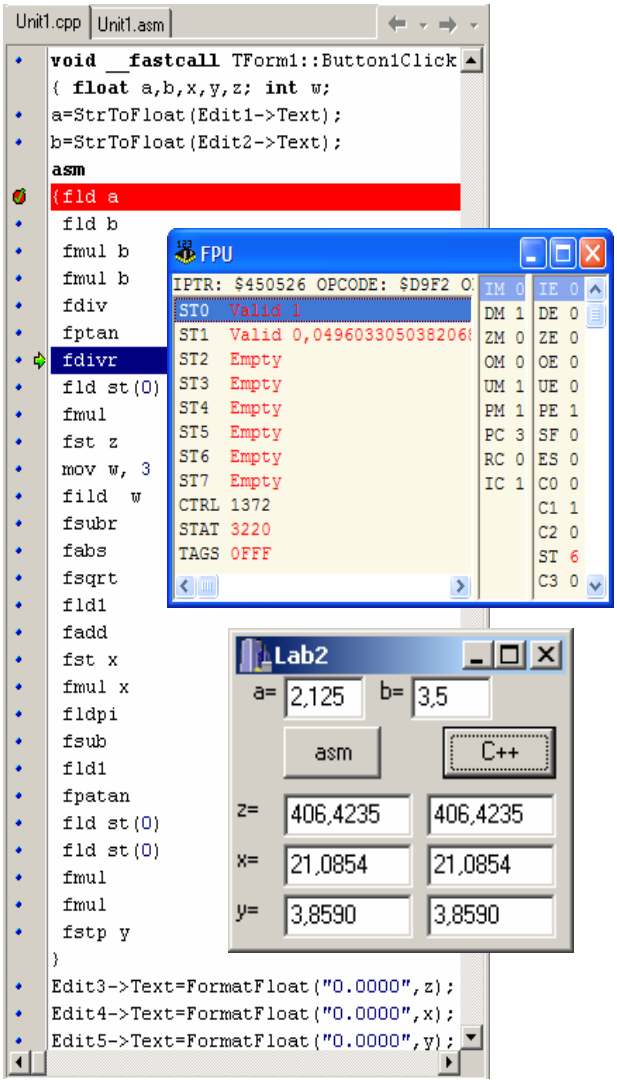
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float a, b, x, y, z;  int w;
  a=StrToFloat(Edit1->Text);
  b = StrToFloat(Edit2->Text);
  asm
  {
   fld a
   fld b
   fmul b
   fmul b
   fdiv
   fptan
   fdivr
   fmul st(0),st(0)
   fst z
   mov w, 3
   fisubr w
   fabs
   fsqrt
   fld1
   fadd
   fst x
   fmul  x
   fldpi
   fsub
   fld1
   fpatan
   fld st(0)
   fld st(0)
   fmul
   fmul
   fstp y
  }
  Edit3->Text=FormatFloat("0.0000",z);
  Edit4->Text=FormatFloat("0.0000",x);
  Edit5->Text=FormatFloat("0.0000",y);
}
```

ST(0)	ST(1)	ST(2)
a		
b	*a*	
b^2	*a*	
b^3	*a*	
a/ b^3		
1	tg*(a/ b^3)*	
ctg*(a/ b^3)*		
ctg^2*(a/ b^3)*		
z=ctg^2*(a/ b^3)*		
z		
3 – z		
|*3 – z*|		
sqrt|*3 – z*|		
1	sqrt|*3 – z*|	
1+sqrt|*3 – z*|		
x=*1*+sqrt|*3 – z*|		
*x*x*		
π	*x*x*	
*x*x*- π		
1	*x*x*- π	
arctan*((x*x–π)/1)*		
arctan*((x*x–π)/1)*	arctan*((x*x–π)/1)*	
arctan*((x*x–π)/1)*	arctan*((x*x–π)/1)*	arctan*(..)*
arctan^2*(...)*	arctan*((x*x–π)/1)*	
arctan^3*(...)*		

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ float a,b,x,y,z;
```

```
  a = StrToFloat(Edit1->Text);
  b = StrToFloat(Edit2->Text);
  z = pow(1/tan(a/(b*b*b)),2);
  x = 1+sqrt(fabs(3-z));
  y = pow(atan(x*x-M_PI),3);
  Edit6->Text=FormatFloat("0.0000",z);
  Edit7->Text=FormatFloat("0.0000",x);
  Edit8->Text=FormatFloat("0.0000",y);
}
```

Порівняємо за розміром коди асемблера та C++ для обчислення функцій з прикладу 2.3. Для цього встановлюємо чотири точки призупину на початку та наприкінці кожного з розв'язань, визначаємо адреси перших та останніх команд та обчислюємо різниці цих адрес. Для асемблерної вставки код першої команди мітиться за адресою 00401BFE, а останньої – 00401C42. Тобто розмір асемблерного коду вставки становить 000401C42 h – 00401BFE h = 44 h = 68 байтів. Обчислення у C++ займає пам'ять від 00401E46 по 00401EDE, тобто 00401EDE h – 00401E46 h = 98 h = 152 байти, що більше ніж удвічі розміру всієї асемблерної вставки. Крім того, розмір виконуваного коду в C++ збільшиться на величину всіх використовуваних функцій математичної бібліотеки math.h: `pow()`, `tan()`, `sqrt()`, `fabs()` тощо. Розміри цих функцій можна визначити аналогічним чином. Наприклад, для функції `tan()`: 32675F48 h – 32675F2C h = 1C h = = 28 байтів; а для функції `pow()`: 2674724h – 32674548 h = 1DCh = 476 байтів. Додавши до розміру основного коду в C++ розмір програмних кодів усіх використаних функцій та порівнявши здобуте число з розміром асемблерної вставки, стає зрозумілим доцільність використання вбудованого асемблера навіть для таких простих програм. Адже аналогічний за результатами, але більш компактний програмний код виконується швидше, що збільшує швидкодію програми.



1.2. ПЕРЕГЛЯД СТАНУ РЕГІСТРІВ CPU ПІД ЧАС ПОКРОКОВОГО ВИКОНАННЯ ПРОГРАМИ

Для покрокового виконання програми й переглядання регістрів співпроцесора слід виконати таку послідовність дій:

– поставити точку переривання у першому рядку асемблерної вставки;

– запустити проект на виконання і ввести вхідні дані у вікна форми, після чого натиснути відповідну командну кнопку, яка запустить програму до першої точки переривання;

– відкрити вікно стану регістрів співпроцесора **View / Debug Windows / FPU** та процесора **View / Debug Windows / CPU**;

– продовжити покрокове виконування програми за допомогою F8 або F7, переглядаючи асемблерні коди програми та вміст усіх регістрів мікропроцесора й співпроцесора.

2. ЛАБОРАТОРНЕ ЗАВДАННЯ

1. Розробити програмний проект у C++ Builder для введення необхідних вихідних даних і розв'язання засобами вбудованого асемблера індивідуального завдання відповідно до варіанта з табл. 2.1. Як елементи керування використовувати командні кнопки: для запускання обчислень, для очищення текстових полів форми і для завершення роботи з програмою. Вивести результати всіх трьох обчислюваних змінних до відповідних текстових полів.

2. Створити таблицю на зразок прикладу 2.3 та заповнити значеннями вмісту використовуваних у проекті регістрів співпроцесора для кожної команди програми. Для заповнення і перевірки значень таблиці у процесі налагодження програми використовувати покрокове виконування команд й відстежувати стан регістрів співпроцесора у вікні дизасемблера.

3. Для зручності перевірки правильності обчислень в асемблері створити додаткову кнопку й написати програмний код для обчислення індивідуального завдання на C++ на зразок прикладу 2.3. Обчислити та порівняти розміри програмного коду в C++ та асемблерної вставки на зразок прикладу 2.3.

4. Оформити протокол лабораторної роботи, в якому, окрім відповідей на відповідні контрольні запитання (вибираються згідно з індивідуальним варіантом на стор. 45) і тексту програми, записати таблицю (див. п. 2) та заповнити її. Занести результати обчислень до протоколу.

Таблиця 2.1. **Варіанти індивідуальних завдань**

№ вар.	Індивідуальне завдання
1	$y = a\sin^2 b + b/\cos a;\ a = \sqrt{\lvert b - c\rvert};\ b = \mathrm{arctg}(x)$
2	$y = \sqrt{a^2 + b^2};\ \ a = \mathrm{tg}\lvert x\rvert;\ \ b = \sin(2 - a)$
3	$y = \mathrm{ctg}(a/c);\ \ c = a^2 + \sqrt{b};\ a = b^3 - \cos\lvert b\rvert$
4	$y = \sqrt{\lvert a - b\rvert};\ \ a = \mathrm{tg}(x);\ \ b = x + \sin(1/x)$

Закінчення табл. 2.1

№ вар.	Індивідуальне завдання
5	$y = a^3 / b^2;\quad a = \sqrt{\lvert x\rvert + 1};\quad b = \sin x^2 - 3$
6	$y = \cos^2(p + \pi);\ p = x^2 - \sqrt{\lvert x\rvert};\quad x = \operatorname{arctg}(1/b)$
7	$y = c^3 / \cos c;\quad c = a^2 + b^2;\quad a = \sqrt{\lvert x - 1\rvert}$
8	$y = \sin^3(a - b);\quad a = t^3 + \sqrt{b};\quad b = \operatorname{tg}^2\lvert x\rvert$
9	$y = \operatorname{arctg}^3 x^2;\quad x = p - k;\quad k = \sqrt{p + t}$
10	$y = \cos^2(a - \sin b);\ a = \sqrt{\lvert x\rvert};\quad b = x^4 + m^2$
11	$y = \sin^3 a + \cos^2 x;\ a = c - k^2;\quad c = \operatorname{arctg}\lvert x\rvert$
12	$y = \cos\sqrt{x - 1};\quad x = a + b^2;\quad a = \sin^2(b - \pi)$
13	$y = a\cos x - b\sin x;\ x = \sqrt{\lvert a - b\rvert};\ a = t^2 b$
14	$y = \sqrt{\lvert x\rvert}\sin a + 5;\quad a = \operatorname{tg}^2\lvert 2x\rvert;\quad x = b/3$
15	$y = 2a/\operatorname{arctg}(b - 1);\quad a = \sqrt{x^2 + b^2};\quad x = \operatorname{tg}(b + \pi)$
16	$y = \cos\lvert x + t^2\rvert;\quad x = t^2 - 4p;\quad t = \operatorname{arctg}(\sqrt{3}/p)$
17	$y = \operatorname{arctg}(1 - a);\quad a = \operatorname{tg}\lvert 4t + b^2\rvert;\quad t = b^2 - \sqrt{3b}$
18	$y = \sqrt{x + 8c};\quad x = \lvert c - a\rvert + a^3;\quad c = \cos^2 a + 1$
19	$y = \sin p + v^3;\quad p = \operatorname{tg}\lvert x\rvert;\quad v = \sqrt{x + 1}/(x^2 - 5)$
20	$y = 4x^3 / t^2;\quad x = \operatorname{tg} 2^a;\quad t = \sin\sqrt{6 - a^3}$
21	$y = c^2 + \sqrt{\lvert a\rvert};\quad c = \operatorname{ctg}(a/x);\quad a = 1 - 8x$
22	$y = \operatorname{arctg}^2\lvert x\rvert;\quad x = \sqrt{t + 8};\quad t = \cos(1 - b)$
23	$y = v + \operatorname{tg}(w - \pi);\quad v = \cos^2 a;\quad w = \sqrt{1 + \lvert 5a\rvert}$
24	$y = x^2 + \sqrt{\lvert 2x - 1\rvert};\quad x = \cos^2 b;\quad b = \sqrt{8 + t^2}$
25	$y = 1 - \cos x^2;\quad x = \operatorname{ctg}(c/a);\quad c = \sqrt{\lvert a - 3\rvert}$
26	$y = \operatorname{ctg}^2\lvert x - \pi\rvert;\quad x = \sqrt{a + b};\quad a = 2^{1+b}$
27	$y = \operatorname{arctg}^3\lvert p - 1\rvert;\quad p = \sqrt{x^2 + 4a};\quad x = \sin(\pi/a)$
28	$y = \sin^2(p + \pi);\quad p = 2^{\sqrt{t}};\quad t = x^2 + \sqrt{\lvert x - 1\rvert}$
29	$y = \cos^3 x + \lvert a\rvert;\quad x = \operatorname{tg}(1/b);\quad b = \sqrt{7 + a} - a^2$
30	$y = \sin(a^2 - 1);\quad a = \sqrt{\lvert b + t\rvert};\quad t = \operatorname{ctg}(b^2 + 1)$

Тема № 4

РОЗГАЛУЖЕННЯ ТА ЦИКЛИ В АСЕМБЛЕРІ

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ЛОГІЧНІ КОМАНДИ АСЕМБЛЕРА

- **AND** *операнд1*, *операнд2*

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

Логічне побітове множення (логічне „і”) операнда1 (регістра чи змінної) та операнда2 (регістра, змінної, числа; операнди не можуть бути змінними водночас). Результат записується в операнд1. Принцип виконання операції AND для різних значень бітів двох операндів подано у таблиці праворуч. Основним використанням цієї операції є обнулення окремих бітів. Наприклад, команда

```
AND AL, 00001111b
```

старші чотири біти регістра AL перетворить на нулі, а молодші залишить без зміни. Аналогічна операція (&) є в мові C++.

- **OR** *операнд1*, *операнд2*

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

Логічне побітове додавання (логічне „або”) операнда1 (регістра чи змінної) й операнда2 (регістра, змінної, числа; операнди не можуть бути змінними водночас). Результат записується в операнд1. Принцип виконання операції OR для різних значень бітів двох операндів подано у таблиці праворуч. Основним використанням цієї операції є встановлення в 1 окремих бітів. Наприклад, команда

```
OR  AL, 00001111b
```

молодші чотири біти регістра AL встановить у 1, а старші залишить без зміни. Аналог команди у C++ – операція |.

- **XOR** *операнд1*, *операнд2*

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

Логічне побітове виняткове „ЧИ” операнда1 (регістра чи змінної) й операнда2 (регістра, змінної, числа; операнди не можуть бути змінними одночасно). Результат записується до операнда1. Принцип виконання операції логічне виняткове „ЧИ” для різних значень бітів двох операндів подано у таблиці праворуч. Тобто біт результату дорівнює 1, якщо відповідні біти операндів розрізняються, і 0 – в інших випадках. Аналогічна операція у C++ – це ^. XOR використовується для різних цілей, наприклад:

```
1)  XOR  AX, AX          // Обнулення регістра AX
або 2)  XOR  AX, BX
        XOR  BX, AX
        XOR  AX, BX      // Обмін вмістом поміж операндами AX та BX
```

Обидва приклади виконуються швидше, аніж відповідні очевидні MOV AX,0 та XCHG AX,BX.

- **NOT** *операнд*

Інверсія кожного з бітів операнда (регістра чи змінної) на протилежний: 0 встановлюється в 1, а 1 – в 0. Аналогічна операція у C++ – це ~.

- **TEST** *операнд1*, *операнд2*

Логічне порівняння двох операндів. Команда TEST виконує ті самі дії, що й логічне AND, проте, *не записує результат у операнд1*, а лише *встановлює прапорці* SF, ZF та PF. Команда TEST, так само як і CMP, використовується переважно у поєднанні з командами умовного переходу.

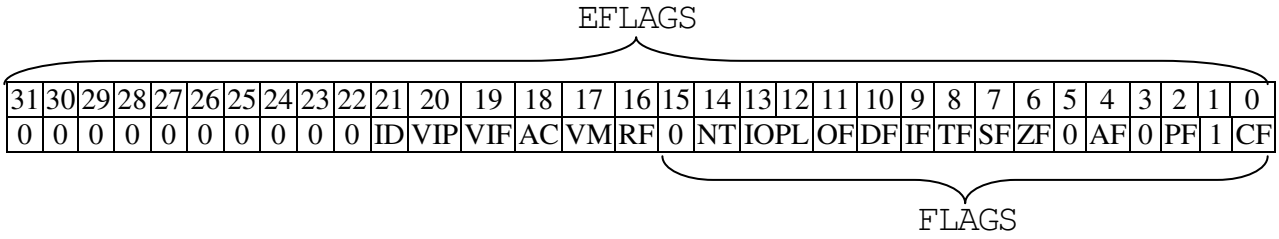
1.2. ПРАПОРЦІ РЕГІСТРІВ FLAGS та EFLAGS

Зупинимось детальніше на структурі регістра прапорців, окремі біти (прапорці) якого мають власну назву і специфічне призначення – зберігання інформації про поточний стан процесора. У 16-розрядних мікропроцесорах цей регістр мав назву FLAGS і мав:

– шість прапорців статусу (біти 0, 2, 4, 6, 7 та 11),
– один прапорець (біт 10) керування станом процесора;
– групу системних прапорців.

При переході до 32- та 64-розрядної архітектури в 32-бітовий регістр EFLAGS та 64-бітовий RFLAGS регістри були додані деякі прапорці, але функції і розміщення старих не змінено. Результатом є так звана "зворотна сумісність" з регістром FLAGS. Це означає, що програми, які використовують FLAGS, можуть без жодних змін працювати на сучасних процесорах.

Регістр EFLAGS має наступну структуру (біти 1, 3, 5, 15, і всі з 22 до 63 зарезервовані для майбутнього використання):



Прапорці статусу (біти 0, 2, 4, 6, 7 і 11) є індикаторами результату роботи арифметичних дій, таких як ADD, SUB, MUL, DIV. Функції прапорців статусу:

– CF (carry flag) – прапорець переносу (біт 0) – встановлюється в 1 ("вмикається"), якщо арифметична операція генерує перенос зі старшого розряду, тобто результат виходить за межі регістра. Показує переповнення при операціях з беззнаковими цілими, а також впливає на деякі операції зсуву (SHR/SAR, SHL/SAL, RCR, RCL);

– PF (parity flag) – прапорець парності (біт 2) – "вмикається", якщо молодший байт результату містить парну кількість одиничок, інакше – "вимикається". Не слід плутати прапорець PF з бітом контролю на парність. Цей прапорець можна використовувати, наприклад, для пошуку помилок при передаванні даних і при виконанні діагностичних тестів;

– AF (auxiliary flag) – прапорець допоміжного переносу (біт 4) – "вмикається", якщо арифметична операція виходить за межі трьох бітів і спричиняє перенос до четвертого біта. Цей прапорець використовується в арифметиці з двійково-десятковими числами (BCD). Двійково-десяткове подання чисел використовується, зокрема, для обміну даними з вимірювальними приладами. Для їх опрацювання в процесорі передбачено цілу низку специфічних команд, при використанні яких доводиться аналізувати стан прапорця допоміжного переносу;

– ZF (carry flag) – прапорець нуля (біт 6) – "вмикається", якщо результат арифметичної команди чи команди порівняння дорівнює нулю, наприклад, якщо від 5 відняти 5 або до 10 додати –10. Команди умовного переходу JE та JZ перевіряють саме цей прапорець;

– SF (sign flag) – прапорець знаку (біт 7) – дорівнює найстаршому бітові результату, який є бітом знаку для знакових цілих: 0 вказує на додатне значення, 1 – на від'ємне;

– OF (owerflow flag) – прапорець переповнення (біт 11) – "вмикається", якщо стався вихід результату за межі припустимого діапазону значень у знаковій цілій арифметиці.

З наведених прапорців лише CF можна змінювати безпосередньо за допомогою інструкцій STC, CLC та CMC. Також, бітові інструкції (BT, BTS, BTR і BTC) копіюють вказаний біт у прапорець CF. Умовні інструкції Jcc (cc – condition code) (перехід за умовою), SETcc (встановлення значення байта-результату залежно від умови), LOOPcc (організація циклу), і CMOVcc (умовне копіювання) використовують ідентифікатори прапорів статусу, як коди умов. Наприклад, інструкція переходу JLE (jump if less or equal – перехід, якщо "менше чи дорівнює" ($\leq$)) перевіряє умови "ZF=1 або SF $\neq$ OF".

Перейдемо тепер до прапорця керування станом процесора DF (direction flag). Цей прапорець (біт 10) використовується командами опрацювання рядків (MOVS, CMPS, SCAS, LODS та STOS) для встановлення напряму: якщо DF = 0, рядки опрацьовуватимуться у порядку зростання адрес (інструкції виконують автодекремент); якщо DF = 1, опрацювання здійснюватиметься у зворотному напрямі (відбувається автоінкремент). Інструкція STD "вмикає" цей прапорець, а CLD – "вимикає". Приклади використання цього прапорця будуть наведені при розгляді відповідних команд процесора.

Системні прапорці регістра EFLAGS контролюють операційну систему та пристрої і не призначені для використання в прикладних програмах:

– TF (trap flag) – прапорець трасування (пастки) (біт 8) – використовується для здійснення покрокового виконування програми. Встановлення цього прапорця дозволяє покроковий режим відлагодження, коли після кожної виконаної інструкції відбувається переривання програми і виклик спеціального оброблювача переривання;

– IF (interrupt enable flag) – прапорець дозволу апаратного переривання (біт 9) – дозволяє (якщо IF = 1) або забороняє (маскує) (якщо IF = 0) процесору реагувати на переривання від зовнішніх пристроїв. Тим самим створюється мо-

жливість виконання найважливіших фрагментів програм без жодних перешкод. Дозвіл і заборона переривань здійснюється спеціальними командами STI (set interrupt – дозволити апаратні переривання) та CLI (clear interrupt – заборонити апаратні переривання);

– IOPL (I/O priviledge level field) – показує рівень пріоритету для операцій введення-виведення виконуваної програми чи задачі (біти 12 і 13). Чим менше рівень, тим більше повноважень має задача. Цей рівень можна модифікувати інструкціями POPF/POPFD і IRET, викликаними на нульовому рівні привілеїв;

– NT (nested task) – прапорець вкладеної задачі (біт 14) – використовується в захищеному режимі роботи процесора для фіксації того факту, що одна задача є кладена в іншу. Встановлюється процесором, коли відбувається перехід від одної задачі до іншої командою виклику підпрограми або апаратним перериванням, і показує, що поточна задача чи то є вкладеною і викликаною з попередньої (прапор NT = 1), чи ні (NT = 0);

– RF (resume flag) – прапорець маскування помилок відлагодження (біт 16) – контролює реакцію процесора на вимкнення відлагодження;

– VM (virtual mode) – віртуальний режим (біт 17) – встановлення цього прапорця в захищеному режимі викликає перемикання в режим віртуального 8086. Для повернення в захищений режим, його “вимикають”;

– AC (alignment check) – перевірка вирівнювання (біт 18) – для ввімкнення перевірки вирівнювання операндів при звертанні до памяті слід встановити цей прапорець і біт AM регістра CR0, оскільки звертання до невирівняних операндів може спричинити виняткову ситуацію. Приміром, Pentium дозволяє розміщувати команди і дані починаючи з довільної адреси. Якщо потрібно контролювати вирівнювання даних і команд за адресами, кратними 2 чи 4, то встановлення цих бітів призведе до того, що всі звертання до некратних адрес спричинюватимуть виняткові ситуації;

– VIF (virtual interrupt flag) – прапорець віртуальних переривань (біт 19) – з’явився в процесорі Pentium і використовується спільно з прапорцем VIP при ввімкнутому розширенні режиму віртуального 8086 для індикації відкладеного переривання;

– VIP (virtual interrupt pending) – прапорець очікування віртуального переривання (біт 20) – встановлюється лише програмно для зазначення наявності відкладеного переривання. Процесором лише зчитується і використовується спільно з прапорцем VIF;

– ID (identification) – прапорець ідентифікації (біт 21). Здатність програми змінити цей прапорець означає підтримку інструкцій CPUID.

Значення деяких прапорців регістра EFLAGS можна змінювати безпосередньо за допомогою спеціальних інструкцій (наприклад, CLD для обнулення прапорця напряму), але не існує інструкцій, які дозволяють звернутися (перевірити чи змінити) до регістра прапорців як до звичайного регістра. Проте, можна зберігати регістр прапорців у стеку або у регістрі EAX і відновлювати регістр прапорців з них за допомогою інструкцій LAHF, SAHF, PUSHF, PUSHFD, POPF і POPFD.

Після того, як вміст EFLAGS записано до стека (або до EAX), прапорці можуть бути відстеженні, і змінені за допомогою інструкцій маніпулювання бітами (BT, BTS, BTR, BTC).

1.3. ДЕЯКІ АСЕМБЛЕРНІ КОМАНДИ ПЕРЕДАЧІ КЕРУВАННЯ

- **JMP** *мітка*

Безумовний перехід на операнд, перед яким встановлено мітку (від англ. jump – стрибок). Розрізняють такі типи переходу:

– short (короткий перехід) – якщо адреса переходу перебуває в межах –128...+127 байт від команди JMP;

– near (ближній перехід) – якщо адреса переходу перебуває у тому самому сегменті пам'яті, що й команда JMP;

– far (дальній перехід) – якщо адреса переходу перебуває у другому сегменті пам'яті.

- **Jcc** *мітка*

Набір команд умовного переходу (типу short чи near) при виконанні певної умови. Перевірка умови здійснюється за рахунок перевірки відповідних прапорців регістра прапорців, які встановлюються попередньою командою, зазвичай CMP або TEST. Основні варіанти команди JCC подано у табл. 4.1.[9]

Таблиця 4.1. **Команди умовного переходу Jcc**

Код	Перевірка прапорців	Позначення умови	Назва умови
JA JBE	ZF=0 та CF=0	> (для чисел без знаку)	якщо вище якщо не нижче і не дорівнює
JG JNLE	ZF=0 та SF=OF	> (для чисел зі знаком)	якщо більше якщо не менше і не дорівнює
JAE JNB JNC	CF=0	≥ (для чисел без знаку)	якщо вище чи дорівнює якщо не нижче якщо немає переносу
JGE JNL	SF=OF	≥ (для чисел зі знаком)	якщо більше чи дорівнює якщо не менше
JB JNAE JC	CF=1	< (для чисел без знаку)	якщо нижче якщо не вище та не дорівнює якщо перенос (англ. *carry*)
JL JNGE	SF≠OF	< (для чисел зі знаком)	якщо менше якщо не більше і не дорівнює
JBE JNA	CF=1 чи ZF=1	≤ (для чисел без знаку)	якщо нижче чи дорівнює якщо не вище

[9] Імена команд побудовані з використанням перших літер відповідних слів англійської мови: *above* (*вище*, для чисел *X* та *Y* вираз *X above Y* означає *X>Y*), *below* (*нижче*, для чисел *X below Y* означає *X<Y*), *equal* (*дорівнює*), *greater* (*більше*), *less* (*менше*), *not* (*невірно, що*). Наприклад, абревіатура JNLE означає "перейти (на мітку), якщо невірно, що *X* менше або дорівнює *Y*".

Закінчення табл. 4.1

Код	Перевірка прапорців	Позначення умови	Назва умови
JBE JNA	CF=1 чи ZF=1	≤ (для чисел без знаку)	якщо нижче чи дорівнює якщо не вище
JLE JNG	ZF=1 чи SF≠OF	≤ (для чисел зі знаком)	якщо менше чи дорівнює якщо не більше
JE JZ	ZF=1	=	якщо дорівнює якщо нуль (англ. *zero*)
JNE JNZ	ZF=0	≠	якщо не дорівнює якщо не нуль
JO	OF=1		якщо є переповнення (англ. *overflow*)
JNO	OF=0		якщо немає переповнення
JP JPE	PF=1		якщо є парність (англ. *parity*) якщо парне
JNP JPO	PF=0		якщо немає парності якщо непарне
JS	SF=1		якщо є знак (англ. *sign*)
JNS	SF=0		якщо немає знаку

Приклади використання команд умовних переходів:

1)
```
    CMP EAX,0
    JNE @M1
    .....
@M1:  INC EAX
```

2)
```
@M2: NEG EAX
     JS  @M2
```

У прикладі 1 виконується перехід на мітку @M1, якщо значення регістра EAX ≠ 0. Приклад 2 дозволяє здобути абсолютне значення числа в EAX, застосовуючи лише дві команди – змінення знаку і перехід на попередню команду повторно, якщо знак є від'ємний.

- **LOOP** *мітка*

Команда циклу, яка виконує зменшення регістра CX на 1 і перехід типу SHORT на мітку, якщо CX ≠ 0. Тобто до регістра CX перед циклом слід занести кількість разів виконування циклу. Наприклад, у поданому нижче фрагменті команда ADD виконується 10 разів і підсумовуються усі числа від 10 до 1, тобто $\sum_{i=1}^{10} i$:

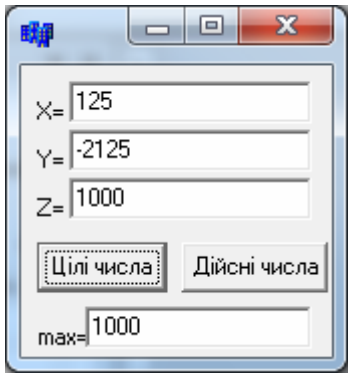
```
            XOR AX, AX
            MOV CX, 0Ah
@Loop_Star: ADD  AX, CX
            LOOP @Loop_Star
```

Команда LOOP еквівалентна до пари команд: DEC CX та JNZ *мітка*, але її код коротше на 1 байт.

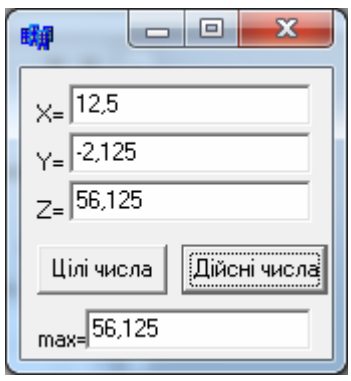
1.4. ПРИКЛАДИ ОРГАНІЗАЦІЇ РОЗГАЛУЖЕНЬ ТА ЦИКЛІВ В АСЕМБЛЕРІ

Приклад 1. Знайти максимальне число з трьох заданих чисел. Наведемо два розв'язки цієї задачі – для цілих та дійсних чисел.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int max, y, z, x = StrToInt(Edit1->Text);
  y = StrToInt(Edit2->Text);  z = StrToInt(Edit3->Text);
  asm
  { mov eax,x       // eax ← x
    mov max,eax     // max ← eax
    cmp eax,y       // Порівняти eax з y
    jg XZ           // Якщо x > y, перейти на порівняння x та z
    mov eax,y       // eax ← y
    mov max,eax     // max ← eax
XZ: cmp eax,z       // Порівняти eax з z
    jg notZ         // Якщо eax > z, перейти
    mov eax,z       // eax ← z
    mov max,eax     // max ← eax
notZ: }
 Edit4->Text=IntToStr(max);
}
```



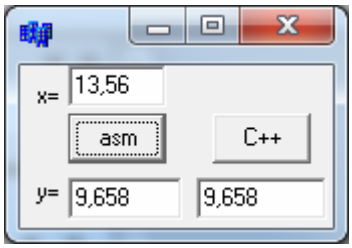

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int max, y, z, x = StrToInt(Edit1->Text);
  y = StrToInt(Edit2->Text);  z = StrToInt(Edit3->Text);
  asm
 {  fld x         // ST(0) ← X
    fst max       // MAX ← ST(0)
    fcomp y       // Порівняти ST(0) з Y, ST(0) порожній
    fstsw ax      // Зберегти регістр стану SR в AX
    sahf          // Завантажити прапорці стану з AH
    ja XZ         // Якщо X > Z, перейти на мітку XY
    fld y         // ST(0) ← Y
    fst max       // MAX ← ST(0)
XZ: fld max       // ST(0) ← MAX
    fcomp z       // Порівняти ST(0) із Z, ST(0) порожній
    fstsw ax
    sahf
    ja notZ       // Якщо ST(0) > Z, перейти на notZ
    fld z         // ST(0) ← Z
    fstp max      // MAX ← ST(0) , ST(0) порожній
notZ:  }
  Edit4->Text=FloatToStr(max);
}
```



Приклад 2. Обчислити суму членів ряду $S=\sum_{k=1}^{7}\frac{k\cos^2(k-x)}{\operatorname{arctg}(k+x^2)}$ для будь-яких значень x.

Для порівняння наведемо два розв'язки цієї задачі – в асемблері та в C++.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ float s,x = StrToFloat(Edit1->Text);
  int k;
 asm
 { mov ecx,7
   mov k,1
   fldz
@next:
   fld x
   fisubr k
   fcos
   fmul st(0),st(0)
   fimul k
   fld x
   fmul x
   fiadd k
   fld1
   fpatan
   fdiv
   fadd
   inc k
   loop @next
   fstp s
 }
 Edit2->Text = FormatFloat("0.000", s);
}
//---------------------------------------------------------
#include "math.h"
void __fastcall TForm1::Button2Click(TObject *Sender)
{ float  s=0, x=StrToFloat(Edit1->Text);
  for(int k=1; k<=7; k++)
    s += k*pow(cos(k-x),2)/atan(k+x*x);
  Edit3->Text = FormatFloat("0.000", s);
}
```



Якщо порівняти розміри програмних кодів, стає зрозуміло, що обчислення у C++ з урахуванням кодів усіх використаних функцій (`pow()`, `cos()`, `atan()`) значно (≈ в 20 разів) більше розміру всієї асемблерної вставки.

2. КОНТРОЛЬНІ ПИТАННЯ

1. Регістр AL містить код 10011011. Яким буде результат команди AND AL,00001111b ? Запишіть цей результат.

2. Регістр AL містить код 10011011. Яким буде результат команди OR AL,00001111b ? Запишіть цей результат.

3. Регістр AL містить код 10011011. Яким буде результат команди XOR AL,00001111b ? Запишіть цей результат.

4. Регістр AL містить код 10011011. Яким буде результат команди NOT AL ? Запишіть цей результат.

5. Яким буде результат команди XOR EAX,EAX ? Відповідь обґрунтуйте.

6. Яким буде вміст регістру EAX після виконання операції XOR AL,AL ? Відповідь обґрунтуйте.

7. Яке призначення регістра прапорців? Яка структура цього регістра? Яких значень може набувати окремий прапорець?

8. Назвіть прапорці статусу. Яким є їхнє призначення?

9. Що таке CF, AF, OF? Поясніть, яким чином вони використовуються.

10. Що таке ZF, SF, PF? Поясніть, в яких випадках і як вони використовуються.

11. Назвіть системні прапорці регістра EFLAGS. В яких випадках і як вони використовуються?

12. Що таке SF? Які команди використовують SF? Наведіть конкретний приклад і поясніть його.

13. Що таке ZF? Які команди використовують ZF? Наведіть конкретний приклад і поясніть його.

14. Що таке PF? Які команди використовують PF? Наведіть конкретний приклад і поясніть його.

15. У чому подібні команди AND та TEST ? А в чому відмінність між ними? Поясніть на прикладах.

16. У чому подібні команди CMP та TEST ? А в чому відмінність між ними? Поясніть на прикладах.

17. У чому подібні команди CMP та SUB ? А в чому відмінність між ними? Поясніть на прикладах.

18. Напишіть фрагмент Assembler-програми, який містив би безумовний перехід. Поясніть, чому у цьому фрагменті потрібен саме такий перехід.

19. Напишіть фрагмент Assembler-програми, який містив би який-небудь умовний перехід. Поясніть, чому у цьому фрагменті потрібен саме такий перехід.

20. Запишіть фрагмент Assembler-програми для визначення максимального з двох чисел X та Y.

21. Як отримати модуль числа за допомогою команд(и) умовного переходу? Запишіть відповідні команди Assembler'а і поясніть їхні дії.

22. Як працює команда `LOOP` ? Поясніть це на прикладі обчислення факторіала.

23. Як організувати цикл, *не користуючись* командою `LOOP` ? Поясніть це на прикладі обчислення факторіала.

24. Які з регістрів процесора відіграють “особливу роль” для організації циклів? Наведіть фрагмент Assembler-програми, який пояснював би цю “особливу роль”.

25. Напишіть Assembler-програму для обчислення значення виразу $1^2+2^2+...+n^2$.

26. Як перевірити, чи містить регістр `ESI` додатне число? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

27. Як перевірити, чи містить регістр `ESI` непарне число? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

28. Як перевірити, чи містить регістр `ESI` число нуль? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

29. Як перевірити, чи містить регістр `ESI` від’ємне число? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

30. Як перевірити, чи містить регістр `ESI` парне число? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

31. Як перевірити, чи містить регістр `ESI` ненульове значення? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

32. Як перевірити, чи містить регістр ESI число в межах [–2, 3]? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

33. Як перевірити, чи містить регістр ESI число, модуль якого не більший за 5 ? Напишіть фрагмент Assembler-програми, що виконує таку перевірку.

34. Напишіть фрагмент Assembler-програми, що виконує перевірку змінної *X* на парність.

35. Напишіть фрагмент Assembler-програми, що виконує перевірку: чи містить змінна *X* від’ємне значення.

36. Напишіть Assembler-програму, що обчислює $\sum_{i=1}^{n} 2^i$.

Лабораторна робота № 3

ОПРАЦЮВАННЯ ЧИСЛОВИХ МАСИВІВ ЗАСОБАМИ ВБУДОВАНОГО В C++ BUILDER АСЕМБЛЕРА

Мета роботи: Вивчення асемблерних умовних і циклічних команд та набуття навичок опрацювання масивів цілих чисел засобами вбудованого в C++ Builder асемблера.

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ЗАСОБИ АСЕМБЛЕРА ДЛЯ ОПРАЦЮВАННЯ МАСИВІВ

При роботі з масивами необхідно пам'ятати, що усі елементи масиву розташовуються в пам'яті комп'ютера послідовно. Однак, саме по собі послідовне розташування даних нічого не говорить про призначення і порядок використання цих даних. І тільки програміст, за допомогою складеного ним алгоритму опрацювання, визначає, як треба трактувати послідовність байтів, що складають масив. Так, одну і ту ж область пам'яті можна трактувати як одновимірний масив, і одночасно ті ж самі дані можуть трактуватися як двомірний масив. Усе залежить тільки від алгоритму опрацювання цих даних у конкретній програмі. Саме тому асемблер не "підозрює" про існування індексів елементів масиву. Коли при програмуванні на асемблері ми говоримо про індекс, то скоріше маємо на увазі не номер елемента в масиві, а певну *адресу*. Наприклад, нехай у C++ оголошено одновимірний масив `int A[10]`, де розмір кожного елемента становить 4 байти. Для доступу до першого елемента цього масиву в асемблері треба просто вказати адресу початку масиву. А для доступу до третього елемента треба до адреси масиву додати сумарну довжину двох його попередніх елементів, тобто 8 байтів. У загальному випадку для обчислення адреси елемента масиву необхідно до початкової (базової) адреси масиву додати добуток розміру елемента масиву на індекс (номер елемента мінус одиниця, оскільки нумеруються елементи з нуля) цього елемента:

база + (*розмір елемента* * *індекс*).

Архітектура мікропроцесора надає досить зручні програмно-апаратні засоби для роботи з масивами. До них відносяться базові та індексні регістри, які дозволяють реалізувати декілька режимів адресації даних:

- *індексна адресація зі зсувом* – режим адресації, при якому ефективна адреса формується з двох складових:

1) постійної (базової) – зазначення прямої адреси масиву у вигляді імені ідентифікатора, що означає початок масиву;

2) змінної (індексної) – зазначення імені індексного регістра.

Наприклад:

```
mov esi,0
mov eax,mas[esi] // Записати 1-й елемент масиву mas у регістр eax:
```

```
inc esi
// Додати до вмісту eax подвійне слово з пам'яті за адресою mas + (esi)*4
add eax,mas[esi*4]
```

Ще один приклад:

```
lea ebx, mas  // Записати в ebx адресу початку масиву mas
// Переслати байт з області даних, адреса якої міститься в регістрі ebx
mov dl,[ebx]
```

➢ *базова індексна адресація зі зсувом* – режим адресації, при якому ефективна адреса формується максимум з трьох складових:

1) постійної (необов'язкова складова), якою може виступати пряма адреса масиву у вигляді імені ідентифікатора, який означає початок масиву, чи то безпосереднє значення;

2) змінної (базової) – зазначення імені базового регістра;

3) змінної (індексної) – зазначення імені індексного регістра.

```
mov ax,mas[ebx][ecx*2] // Адреса операнда дорівнює [mas+(ebx)+(ecx)*2]
. . .
sub dx,[ebx+8][ecx*4]  // Адреса операнда дорівнює [(ebx)+8+(ecx)*4]
```

Тут для опису адреси використовуються два регістри, тобто йдеться про базово-індексну адресацію. Лівий регістр розглядається як базовий, а правий – як індексний. У загальному випадку це не є принциповим, але якщо використовується *масштабування* (див. далі) з одним із регістрів, то він завжди є індексним.

Слід зауважити, що базово-індексну адресацію не забороняється поєднувати з прямою адресацією чи зазначенням безпосереднього значення. Адреса тоді формуватиметься як сума усіх складових.

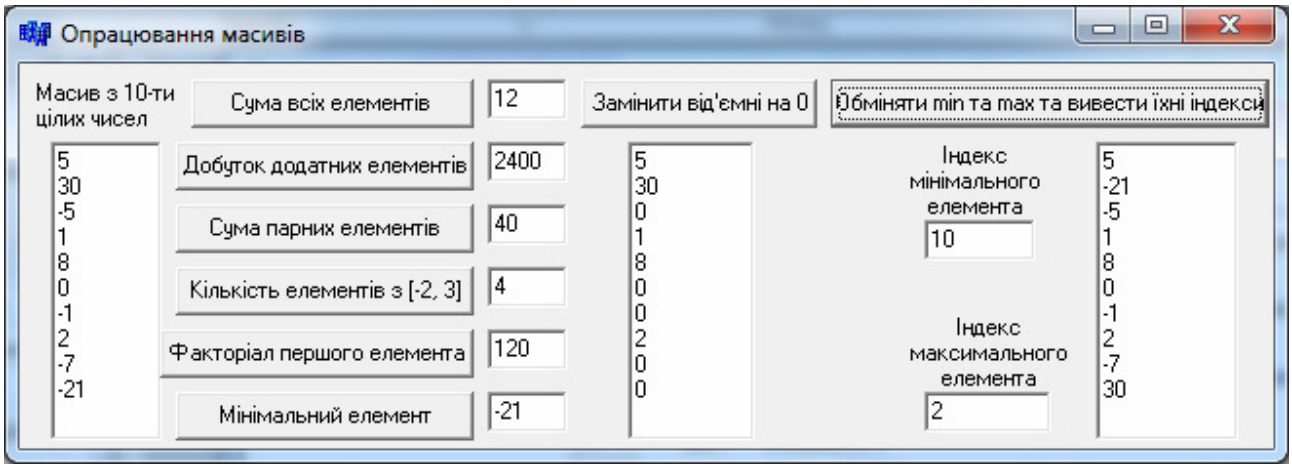
Нагадаємо, що як базовий регістр може використовуватися будь-який з восьми регістрів загального призначення. В якості індексного регістра також можна використовувати будь-який регістр загального призначення, за винятком esp/sp.

Мікропроцесор дозволяє *масштабувати* індекс. Це означає, що якщо вказати після імені індексного регістра знак множення * з подальшою цифрою 2, 4 або 8, то вміст індексного регістра множитиметься на 2, 4 або 8, тобто масштабуватися. Застосування масштабування полегшує роботу з масивами, які мають розмір елементів, рівний 2, 4 або 8 байт, оскільки мікропроцесор сам проводить корекцію індексу для отримання адреси чергового елемента масиву. Нам треба лише завантажити в індексний регістр значення необхідного індексу (нумерація індексів розпочинається з 0):

```
mov esi,0              // Індекс в esi
mov eax,mas[esi*4]     // Перший елемент масиву в eax
inc esi                // На наступний елемент
```

1.2. ПРИКЛАДИ ОПРАЦЮВАННЯ ОДНОВИМІРНИХ МАСИВІВ У АСЕМБЛЕРІ

Приклад 3.1. В цьому прикладі у програмному проекті показано шість найбільш поширених алгоритмів опрацювання елементів одновимірного масиву з 10 цілих чисел.




```
// Сума всіх елементів масиву
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i, s, a[10];
  for(i=0; i<10; i++) a[i] = StrToInt(Memo1->Lines->Strings[i]);
  asm
  { mov edx, 0                  // Початковий зсув
    mov eax, 0                  // На початку сума = 0
    mov ecx, 10                 // Кількість елементів масиву для циклу loop
@n1: add eax, dword ptr a+edx   // Додати до суми (eax) новий елемент масиву
    add edx, 4                  // Обчислити зсув для наступного елемента
    loop @n1                    // Цикл для опрацювання усіх елементів
    mov s, eax                  // Вивантажити результат
  } Edit1->Text= IntToStr(s);
}  //---------------------------------------------------------
// Добуток додатних елементів масиву
void __fastcall TForm1::Button2Click(TObject *Sender)
{ int i, p, k, a[10];
 for(i=0; i<10; i++) a[i]=StrToInt(Memo1->Lines->Strings[i]);
  k = sizeof(a) / 4;            // Кількість елементів масиву
asm
 { mov edi, 0                   // Початковий зсув
   mov eax, 1                   // Початкове значення добутку в EAX = 1
   mov ecx, k                   // Кількість повторень циклу
 @n2:
   mov esi, dword ptr a+edi     // Завантажити esi в черговий елемент масиву
   cmp esi, 0                   // Порівняти цей елемент з 0
   jle @net                     // Якщо не від'ємний,
   imul esi                     // помножити на eax
```

```
 @net:
   add edi, 4                    // Підготувати зсув для наступного елемента
   loop @n2              // Цикл: ecx=ecx–1 і, якщо ecx>0, перейти на мітку @n2
   mov p, eax                    // Вивантажити результат
  } Edit2->Text = IntToStr(p);
} //---------------------------------------------------------
// Сума парних елементів масиву
void __fastcall TForm1::Button3Click(TObject *Sender)
{ int k, i, s, a[10];
 for(i=0; i<10; i++)
    a[i] = StrToInt(Memo1->Lines->Strings[i]);
 k = sizeof(a)/4; // Кількість елементів масиву
 asm
{  pusha      // Запам'ятовуємо в стеку 8 регістрів: AX, CX, DX, BX, SP, BP, SI, DI –
              // з метою їх відновлення командою popa після виконання
              // даної програми, що є правилом хорошого тону
   mov edx,0 // Зсув першого елемента відносно адреси початку цього масиву = 0
   mov eax,0 // Початкове значення суми = 0
   mov ecx,k // Кількість повторень циклу
@m2:          // Мітка початку циклу
   mov esi, dword ptr a+edx // Переслати в ESI подвійне слово (4 байти),
              // взявши його з адреси: початок масиву + зсув (EDX), тобто
              // відбувається завантаження вмісту чергового елемента масиву в циклі
   and esi,1 // Побітове логічне множення на 1. Те саме, що and esi, 0...01b. Тобто
              // молодший біт в ESI множиться логічно на 1, а решта бітів – на 0.
              // Якщо в ESI було непарне число, результатом у ESI буде 1, інакше – 0.
   jnp @m1    // Якщо результат останньої виконаної перед jnp команди був
              // НЕпарним (PF=0), перейти до @m1. Отже, перехід НЕ відбудеться,
              // якщо попередній вміст ESI був парним чи нулем.
   add eax, dword ptr a+edx // Додати до суми (EAX) значення чергового
              // парного елемента масиву, доступ до якого здійснюється за адресою:
              // початок масиву + зсув (EDX)
@m1:          // Мітка переходу для непарних елементів
   add edx,4 // Збільшуємо зсув (EDX) на 4 для наступного елемента масиву
   loop @m2  // Цикл: ecx=ecx–1  і, якщо ecx>0, перейти на мітку @m2
   mov s, eax // Записати накопичену в акумуляторі суму в змінну s.
   popa       // Відновити вміст 8-ми регістрів AX, CX, DX, BX, SP, BP, SI, DI
  }
 Edit3->Text = IntToStr(s);
}
//---------------------------------------------------------
// Кількість елементів з діапазону [-2, 3]
void __fastcall TForm1::Button4Click(TObject *Sender)
{ int i, kol, a[10];
  for(i=0; i<10; i++) a[i]=StrToInt(Memo1->Lines->Strings[i]);
```

```
  asm
 { pusha
   mov edi, 0                    // Початковий зсув
   mov eax, 0                    // На початку кількість = 0
   mov ecx, 10                   // Кількість повторень циклу
@nex: mov esi, dword ptr a+edi   // Завантажити в esi черговий елемент масиву
   cmp esi, -2                   // Порівняти цей елемент з  -2
   jl @not                       // Якщо не менше  –2,
   cmp esi, 3                    // порівняти з 3,
   jg @not                       // і, якщо не більше 3,
   inc eax                       // збільшити значення кількості на 1
@not: add edi,4                  // Підготувати зсув для наступного елемента
   loop @nex                     // Цикл: якщо ecx>0, перейти на мітку
   mov kol, eax                  // Вивантажити результат
   popa
  }
 Edit4->Text = IntToStr(kol);
}
//---------------------------------------------------------
// Факторіал першого елемента масиву
void __fastcall TForm1::Button5Click(TObject *Sender)
{ int i, f, a[10];
  for(i=0; i<10; i++)
    a[i]=StrToInt(Memo1->Lines->Strings[i]);
  asm
  { mov eax, 1                   // На початку факторіал в eax = 1
    mov ecx, dword ptr a         // Для 10-го елемента слід записати (a+4*9)
@next:
    imul ecx                     // Домножити eax на ecx
    loop @next
    mov f, eax
  }
  Edit5->Text = IntToStr(f);
}
//---------------------------------------------------------

// Мінімальний елемент
// Наведемо 2 можливі варіанти розв'язання цієї задачі, які відрізняються
// засобами адресації елементів та командами порівняння
void __fastcall TForm1::Button6Click(TObject *Sender)
{ int i, min, a[10];
  for(i=0; i<10; i++)
    a[i]=StrToInt(Memo1->Lines->Strings[i]);
```

```
  asm
 {
 mov ecx,10 // Кількість елементів
 dec ecx
 lea edx,[a+ecx*4]
 neg ecx
 mov eax,[edx]
@b:
 cmp eax,[edx+ecx*4]
 cmovg eax,[edx+ecx*4]
 inc   ecx
 jne @b
 mov min,eax
  }
 Edit6->Text = IntToStr(min);
}
```

```
 mov esi,dword ptr a
 mov min,esi // min = a[0]
 mov edi,4    // Зсув на a[1]
 mov ecx,9//Лишилося перевірити 9 елементів
@nn:
 mov esi,dword ptr a+edi
 cmp esi,min
 jg @nt
 mov min,esi
@nt: add edi,4
 loop @nn
```

// Використана команда CMOVG є командою умовного пересилання даних, яка пересилатиме дані
*// з другого операнда [edx+ecx*4] до першого операнда eax за умови, що вміст першого операнда*
// більше другого. Взагалі, команда CMOVcc може бути сформована з будь-якою умовою
// аналогічно до команди Jcc (див. табл. 4.1, тема 4).

```
//---------------------------------------------------------
```

// Замінити всі від'ємні елементи на нулі

```
void __fastcall TForm1::Button7Click(TObject *Sender)
{ int i, a[10];
  for(i=0; i<10; i++) a[i] = StrToInt(Memo1->Lines->Strings[i]);
  asm
 {  mov edx, 0                  // Початковий зсув
    mov ecx, 10                 // Кількість повторень циклу
@next:
    mov esi, dword ptr a+edx    // Завантажити черговий елемент в esi
    cmp esi, 0                  // Порівняти елемент з 0
    jge @net                    // Якщо елемент ≥ 0, перейти на мітку
    mov dword ptr a+edx,0       // Записати замість елемента 0
@net: add edx, 4                // Підготувати зсув для наступного елемента
    loop @next                  // Цикл: якщо ecx>0, перейти на мітку
  }
  Memo2->Clear();
  for(i=0; i<10; i++) Memo2->Lines->Add(IntToStr(a[i]));
}
//---------------------------------------------------------
```

// Поміняти місцями мінімальний та максимальний елементи та вивести їхні індекси

```
void __fastcall TForm1::Button8Click(TObject *Sender)
{ int i, A[10], IndMin, IndMax;
  for(i=0; i<10; i++) A[i] = StrToInt(Memo1->Lines->Strings[i]);
```

```
  asm
 { // Пошук індексів мінімального та максимального елементів
   mov IndMax,0        // IndMax=0
   mov IndMin,0        // IndMin=0
   mov esi, A[0]       // В ESI шукатимемо максимальний елемент, спочатку ESI=A[0]
   mov edi, A[0]       // В EDI шукатимемо мінімальний елемент, спочатку EDI=A[0]
   mov ecx, 9          // Кількість елементів, які лишилося перевірити
M1: mov eax, A[ecx*4]// Завантажити в EAX черговий елемет масиву
   cmp eax, esi        // Порівняти його з максимальним (ESI) і,
   jle M2              // якщо він не менше і не дорівнює, тобто більше,
   mov esi, eax        // зберегти його значення в ESI та
   mov IndMax, ecx     // запам'ятати індекс в IndMax.
M2: cmp eax, edi       // Порівняти елемент з мінімальним (EDI) і,
   jge M3              // якщо він не більше і не дорівнює, тобто менше,
   mov edi, eax        // зберегти його значення в EDI та
   mov IndMin, ecx     // запам'ятати індекс в IndMin.
M3: loop M1            // Цикл: якщо ecx>0, перейти на мітку
   // Обмін місцями мінімального та максимального елементів
   mov ecx,IndMax      // Завантажити в ECX індекс максимального елемента
   shl ecx,2           // Перетворити індекс на зсув, помноживши його на 4 (розмір int)
   mov dword ptr A+ecx,edi// Замінити максимальний елемент на значення мінімального
   mov ecx,IndMin      // Завантажити в ECX індекс мінімального елемента
   shl ecx,2           // Перетворити індекс на зсув, помноживши його на 4 (розмір int)
   mov dword ptr A+ecx,esi// Замінити мінімальний елемент на значення максимального
   inc IndMax          // Відкорегувати індекси,збільшивши їх на 1, оскільки
   inc IndMin          // нумерація елементів в масиві розпочинається з 0, а не з 1.
  }
  Memo3->Clear();
  for(i=0; i<10; i++) Memo3->Lines->Add(IntToStr(A[i]));
  Edit7->Text=IntToStr(IndMin);
  Edit8->Text=IntToStr(IndMax) ;
 }
```

Приклад 3.2. У масиві з 8-ми цілих чисел визначити індекс першого нуля. Якщо ж масив не містить нулів, повернути число -1 як результат.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ const int LenA=8;     int k, i, ind, A[LenA];
  for(i=0; i<LenA; i++) A[i]=StrToInt(Memo1->Lines->Strings[i]);
  const int size_int=sizeof(int);
  k=sizeof(A)/size_int; // Кількість елементів
  asm
  { pusha
    mov edx, 0    // Зсув від початку масиву
    mov eax, 0    // EAX збільшуватиметься щоразу на 1 і відповідатиме індексу елемента
    mov ecx, k
```

```
@next:
   mov esi, dword ptr A+edx // Завантажити в ESI черговий елемент масиву
   cmp esi,0       // та порівняти його з нулем і, якщо
   je @Zero        // елемент має нульове значення, перейти на мітку @Zero
   inc eax         // Індекс поточного елемента масиву
   add edx, size_int // Збільшити зсув на довжину одного елемента
   loop @next
@Zero:
   // Якщо нулів у масиві нема, на останньому кроці в EAX буде значення довжини рядка LenA
   cmp eax, LenA // Перевірка наявності нулів, тобто збігу значень EAX та LenA
   jne @Fin    // Якщо значення не збігаються, тобто нулі були, перейти на мітку @Fin,
   mov eax, -1    // інакше завантажити число -1 в  EAX
@Fin:
   mov ind, eax
   popa
  }
  Edit1->Text= IntToStr(ind);
}
```

2. ЛАБОРАТОРНЕ ЗАВДАННЯ

1. Розробити програмний проект C++ Builder для введення елементів масиву та розв'язання засобами вбудованого асемблера індивідуального завдання відповідно до варіанта з табл. 3.1.

2. При відлагодженні програми слід використовувати покрокове виконування команд і відстежувати стан регістрів мікропроцесора у вікні дизасемблера.

3. Для зручності перевірки правильності обчислень в асемблері створити додаткову кнопку й написати програмний код для обчислення індивідуального завдання мовою C++.

4. Оформити протокол лабораторної роботи, до якого занести відповіді на відповідні контрольні запитання (вибираються згідно з індивідуальним варіантом на стор. 64) і записати тексти програм. Занести результати обчислень до протоколу.

Таблиця 3.1. **Варіанти індивідуальних завдань**

№ вар.	Розмір масиву	Індивідуальне завдання
1	15	Обчислити суму від'ємних елементів масиву
2	10	Обчислити кількість додатних елементів масиву
3	8	Обчислити факторіал значення останнього елемента масиву
4	12	Обчислити добуток елементів, значення яких є менше за 6
5	14	Обчислити суму непарних елементів масиву
6	18	Обчислити суму елементів, абсолютне значення яких не перевищує 5

№ вар.	Розмір масиву	Індивідуальне завдання
7	11	Обчислити кількість від’ємних елементів масиву

Закінчення табл. 3.1

№ вар.	Розмір масиву	Індивідуальне завдання
8	14	Обчислити факторіал значення третього елемента масиву
9	16	Знайти індекс мінімального елемента
10	14	Обчислити кількість елементів масиву, значення яких є більше за значення першого елемента
11	17	Обчислити суму мінімального й максимального елементів масиву
12	9	Обчислити суму елементів, значення яких перебувають у діапазоні [3, 6]
13	15	Знайти максимальний елемент масиву
14	10	Обчислити добуток непарних елементів масиву
15	8	Обчислити суму мінімального елемента масиву та його індексу
16	12	Обчислити різницю поміж мінімальним і максимальним елементами масиву
17	20	Обчислити кількість парних елементів масиву
18	18	Обчислити суму квадратів тих чисел, модуль яких не перевищує 3
19	11	Обчислити факторіал значення сьомого елемента масиву
20	9	Обчислити добуток від’ємних елементів масиву
21	16	Обчислити суму мінімального елемента масиву та його індексу
22	19	Обчислити кількість додатних та від’ємних елементів масиву
23	17	Обчислити добуток елементів, значення яких перебувають в діапазоні [2, 5]
24	8	Обчислити суму елементів, значення яких є менше за значення першого елемента масиву
25	7	Обчислити добуток ненульових елементів масиву
26	18	Обчислити суму додатних елементів, значення яких не перевищує 4
27	11	Обчислити добуток мінімального й максимального елементів масиву
28	10	Обчислити суму модулів усіх від’ємних елементів масиву
29	12	Обчислити різницю поміж максимальним та мінімальним елементами масиву
30	9	Обчислити добуток елементів, значення яких є менше за значення останнього елемента масиву

Тема № 5

ОПРАЦЮВАННЯ РЯДКІВ ЗАСОБАМИ АСЕМБЛЕРА

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. ДЕЯКІ АСЕМБЛЕРНІ КОМАНДИ РОБОТИ З РЯДКАМИ ТА БЛОКАМИ ДАНИХ

Ці команди використовують для певних операцій над рядками (тобто неперервними послідовностями) байтів, слів та подвійних слів. За умовчанням процесор використовує регістри `DS:SI`, щоб задати адресу вхідного рядка, і регістри `ES:DI`, щоб вказати рядок-результат.[10] Програміст повинен забезпечити правильні початкові значення цих регістрів. При виконанні команд опрацювання рядків значення регістрів `SI` та/або `DI` автоматично змінюється після опрацювання кожного елемента рядка – так, щоб відповідна адреса вказувала на елемент, який слід обробити на наступному кроці. А саме, значення `SI` та/або `DI` збільшується чи зменшується залежно від:

a) прапорця `DF` (`DF=1` – збільшення, `DF=1` – зменшення значення);

b) *крок* зміни значення дорівнює 1 при опрацюванні байтів, 2 при опрацюванні слів, і 4 при опрацюванні подвійних слів.

- **MOVS** *операнд*, *рядок* // *Копіювання рядка на адресу операнда*

```
MOVSB          // Копіювання рядка байтів з [DS:SI] в [ES:DI]
MOVSW          // Копіювання рядка слів з [DS:SI] в [ES:DI]
MOVSD          // Копіювання рядка подвійних слів з [DS:SI] в [ES:DI]
```

- **LODS** *рядок* // *Зчитування з рядка в акумулятор (AL, AX чи EAX)*

```
LODSB          // Зчитування одного байта з рядка в акумулятор (AL)
LODSW          // Зчитування одного слова з рядка в акумулятор (AX)
LODSD          // Зчитування подвійного слова з рядка в акумулятор (EAX)
```

- **STOS** *рядок* // *Записування значення акумулятора (AL, AX чи EAX) до рядка*

```
STOSB          // Записування значення акумулятора (AL) до рядка
STOSW          // Записування значення акумулятора (AX) до рядка
STOSD          // Записування значення акумулятора (EAX) до рядка
```

- **INS** *змінна*, `DX` // *Зчитування рядка з порту (номер порту в DX) у змінну*

```
INSB // Зчитування байта з порту (номер порту в DX) в пам'ять на адресу [ES:DI]
INSW // Зчитування слова з порту (номер порту в DX) в пам'ять на адресу [ES:DI]
INSD // Зчитування 2-х слів з порту (номер порту в DX) в пам'ять на адресу [ES:DI]
```

---

[10] Зверніть увагу: якщо якась з обговорюваних команд працює з *одним* рядком, то цей рядок вважається вхідним або вихідним у залежності від *типу* команди. Наприклад, команди типу **LODS** вважають рядок, елементи якого завантажуються в акумулятор, *вхідним* (і використовують відповідно адресу рядка з `DS:SI`). А команди типу **SCAS** вважають рядок, елементи якого порівнюються з вмістом акумулятора, *вихідним* (і використовують адресу рядка з `ES:DI`).

- **INS** *змінна*, DX // *Зчитування рядка з порту (номер порту в DX) у змінну*

INSB // *Зчитування байта з порту (номер порту в DX) в пам'ять на адресу [ES:DI]*
INSW // *Зчитування слова з порту (номер порту в DX) в пам'ять на адресу [ES:DI]*
INSD // *Зчитування 2-х слів з порту (номер порту в DX) в пам'ять на адресу [ES:DI]*

- **OUTS** DX, *змінна* // *Записування рядка до порту (номер порту в DX) змінної*

OUTSB // *Записування до порту (номер порту в DX) байта пам'яті з адресою [DS:SI]*
OUTSW // *Записування до порту (номер порту в DX) слова пам'яті з адресою [DS:SI]*
OUTSD // *Записування до порту (номер порту в DX) 2-х слів з адреси [DS:SI]*

- **CMPS** *операнд*, *рядок* // *Порівняння операнда з рядком і встановлення прапорців*

CMPSB // *Порівняння рядка байтів із [DS:SI] з [ES:DI]*
CMPSW // *Порівняння рядка слів із [DS:SI] з [ES:DI]*
CMPSD // *Порівняння рядка подвійних слів із [DS:SI] з [ES:DI]*

- **SCAS** *рядок* // *Порівняння акумулятора (AL, AX чи EAX) із рядком*

SCASB // *Порівняння AL з рядком і встановлення прапорців*
SCASW // *Порівняння AX з рядком і встановлення прапорців*
SCASD // *Порівняння EAX з рядком і встановлення прапорців*

- **REP** // *Повторювати наступну команду. Число повторень задається у*
// *регістрі CX (ECX) і зменшується на 1 після кожного повтору*

REPE // *Повторювати наступну команду, допоки прапорець ZF=0*
REPNE // *Повторювати наступну команду, допоки прапорець ZF≠1*
REPZ // *Повторювати наступну команду, допоки прапорець ZF=0*
REPNZ // *Повторювати наступну команду, допоки прапорець ZF≠1*

Будь-яка з команд останньої групи вживається як *префікс* до команд опрацювання рядків, наприклад:

```
REP MOVSW
```

Застосування префікса спричиняє повторювання команди опрацювання рядків. Повторювання буде припинено, коли регістр CX (ECX) міститиме число 0. Крім того, REPE та REPZ припиняють повторення команди, якщо прапорець ZF=0, а REPNE і REPNZ припиняють повторення, якщо прапор ZF=1. Префікс REP, зазвичай, використовується з командами MOVS, LODS, STOS, INS та OUTS, а префікси REPE, REPZ, REPNE та REPNZ – з командами CMPS та SCAS.

Команди опрацювання рядків MOVS, CMPS, SCAS, LODS та STOS використовують прапорець керування станом процесора DF (Direction Flag) для встановлення напрямку "пересування" вздовж рядка: якщо DF=0, рядки опрацьовуватимуться у порядку зростання адрес (інструкції виконують автоінкремент); якщо DF=1, опрацювання здійснюватиметься у зворотному напряму (відбувається автодекремент). Інструкція STD "вмикає" прапорець DF (після STD матимемо DF=1), а CLD – "вимикає".

1.2. ТИПИ РЯДКІВ У C++ BUILDER

При виконуванні програм комп'ютер витрачає, за деякими оцінками, до 70 % часу на маніпулювання з текстовими рядками: копіює їх з одного місця

пам’яті в інше, перевіряє наявність у рядках певних слів, поєднує чи відсікає рядки тощо. Щоб вживати команди асемблера для опрацювання текстового рядка, оголошеного засобами мови високого рівня, необхідно знати, який формат збереження рядка передбачено відповідним оголошенням. У цьому параграфі ми стисло нагадаємо відомості про основні формати символьних рядків у “стандарті” мови С, та у “діалекті” цієї мови С++-Builder.

Рядки “стандартного” С.

Майже всі різновиди рядків у С становлять собою послідовність (масив) символів із завершальним нуль-символом. Нуль-символ (нуль-термінатор) – це символ з кодом 0,[11] у текстах мови С він записується у вигляді послідовності '\0'. За розташуванням нуль-символу визначається фактична довжина рядка.

Доступ до рядка може здійснюватись через вказівник на перший символ рядка. Значенням змінної-імені рядка у С є адреса першого символу цього рядка.

Рядок може бути оголошено як масив символів або як змінну типу **`char*`**. Наведемо два приклади еквівалентних оголошень С++-рядків з 9-ти символів:

```
char S[] = "Приклад 1";
char  *S = "Приклад 1";"
```

Будь-яке з цих оголошень спричиняє наступні значення елементів рядка S: `S[0]='П'`, `S[1]='р'`, … , `S[5]='а'`, `S[6]='д'`, `S[7]=' '`, `S[8]='1'`, S[9]= '\0'.

Символи можна порівнювати. Більшим вважається символ з більшим двійковим кодом, тобто символ, розташований у таблиці ANSI-кодів пізніше. Наприклад: `'a'<'h'`, `'A'<'a'`.

Оскільки символи кодуються двійковими числами, змінні типу `char` у мові С можна додавати й віднімати. Результатом додавання буде символ, код якого дорівнює сумі кодів символів-доданків, наприклад:

```
char c = 'A';
char c1 = c + 5;    // c1 = 'F'
char c2 = c + 32;   // c2 = 'a'
char c3 = c – 10;   // c2 = '7'
```

Проте слід пам’ятати, що такі операції над символами вважають `char`-змінні 1-байтовими цілими числами зі знаком. Отже, якщо код символа є більшим за 127 (тобто старший біт відповідного байта – це 1), у операціях додавання/віднімання символ вважатиметься *від’ємним* числом. Наприклад, оскільки літера кирилиці `'я'` – має код 11111111, що відповідає цілому числу -1, після операції:

```
char c4 = c + 'я';
```

отримаємо `c4 = '@'` (символ ANSI-таблиці, який безпосередньо передує символу `'A'`).

---

[11] Нагадаємо: кожен символ (величина типу `char`) займає 1 байт, а *код* символу – це двійкове число, записане у цей байт. Таблиця, яка встановлює відповідність між символами та їхніми двійковими кодами, у системі Windows має назву “ANSI-таблиця”.

Рядки C++ Builder.

Для опрацювання текстів у C++ Builder є "додаткові", порівняно з "стандартним" C, типи рядків. Далі наведено три таких типи. Для будь-якого з них рядок вважається одновимірним масивом символів, тобто до символів рядка можна звертатися за їхніми індексами у цьому рядку. Проте, *на відміну від "стандартного" C, індексація символів рядка починається з 1*.

1. "Короткий" рядок **SmallString**<n>. При оголошенні змінної цього типу для неї виділяється n+1 байт (по одному байту на кожний символ). Наприклад, при оголошенні

```
SmallString<10> S;
```

за змінною S закріплюється 11 байтів. Перший байт (вважається, що він має індекс 0) містить значення поточної довжини рядка. Отже, якщо виконати оператор S="C++"; матимемо S[0]=4, S[1]='C', S[2]='+', S[3]='+', а решта 7 закріплених за S байтів міститиме випадкові двійкові числа (так зване "сміття").

Зрозуміло, що в оголошенні "короткого" рядка $1 \le n \le 255$. У випадку $n = 255$ можна вживати особливий зарезервований тип ShortString (тобто оголошення ShortString S еквівалентне до SmallString<255> S):

```
ShortString S="Приклад 2";
```

Тут S[0]=9, S[1]='П', S[2]='р', … , S[6]='а', S[7]='д', S[8]=' ', S[9]='2', а наступні 255–9=246 закріплених за S байтів містять "сміття".

Значенням змінної, оголошеної як SmallString<n> або ShortString, є *адреса* нульового байта (байта довжини) цієї змінної. Враховуючи це, зрозуміло, що наступна команда асемблера завантажує байт за адресою оголошеного вище рядка S у регістр al. Отже, al міститиме число, рівне довжині рядка:

```
mov al, byte ptr[S]        // al = 9
```

2. "Довгий" рядок **AnsiString** з нульовим символом '\0' наприкінці (нуль-термінальний рядок).[12] У результі оголошення

```
AnsiString S;
```

у змінну S буде записано *адресу* певної області пам'яті, а за цією адресою буде розміщено символ '\0' (отже, змінні цього типу ініціюються порожніми рядками).

На відміну від рядків типу char*, індексація символів в AnsiString-рядках починається з 1.[13] Наведемо приклад:

```
AnsiString S="Приклад 3";
```

Тут матимемо: S[1]='П', S[2]='р', … , S[6]='а', S[7]='д', S[8]=' ', S[9]='3', S[10]='\0'.

---

[12] В оголошеннях допускається вживати синонім для імені цього типу – String.

[13] Це зумовлено тим, що цей тип запозичено Borland C++ Builder з мови програмування Borland Delphi (Pascal).

3. “Широкий” рядок – **WideString** – з нульовим символом '\0' наприкінці. Цей тип подібний до AnsiString, проте використовує *два* байти пам’яті для розміщення одного символа. У результаті число можливих символів зростає до $2^{16} = 65\,536$.

Змінні типів AnsiString чи WideString – це динамічно розміщувані у пам’яті масиви символів, а їхня максимальна довжина обмежується лише наявністю пам’яті (так, AnsiString-рядок може містити до $2^{32}–1$ символів).

Усі три наведені типи рядків в C++ Builder зручно пристосовані до введення та виведення рядків за допомогою компонентів форми: текстові поля (властивості) компонентів узгоджуються зі згаданими типами. Наприклад, якщо змінна S оголошена як SmallString, чи AnsiString, чи WideString, – введення та виведення рядка S через компонент Edit1 можна подати у той самий спосіб:

```
S=Edit1->Text;      // Введення рядка з Edit1
Edit2->Text=S;      // Виведення рядка до Edit2
```

Для введення з компонентів форми рядків “стандартного” C слід застосувати функцію (метод) перетворення типів c_str():

```
char *s="";
strcpy(s, Edit1->Text.c_str() );
```

Відповідно для виведення рядка “стандартного” C у компоненти форми застосовується функція AnsiString(char*), наприклад:

```
char *s = "Рядок стандартного C";
Edit1->Text = AnsiString(s);
```

1.3. ОРГАНІЗАЦІЯ АСЕМБЛЕРНИХ ФУНКЦІЙ

C++ Builder підтримує можливість організації функцій (підпрограм) з асемблерними командами. При організації таких функцій слід дотримуватись усіх відповідних правил мови C++.

Зверніть увагу, що якщо деяка функція C++ повертає результат R типу int або char, оператор return R заносить цей результат у регістр-акумулятор (eax). Отже, якщо асемблерна вставка у підпрограму “сама” запише її результат в eax, команда return стає не обов’язковою. Наприклад, функція обчислення суми двох чисел може мати вигляд:

```
int  sum( int a, int b)
{ asm
  { mov eax, a
    add eax, b
} }
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int a = StrToInt(Edit1->Text), b = StrToInt(Edit2->Text);
  int c = sum(a, b);
  Edit3->Text = IntToStr (c);
 }
```

Функція (підпрограма) визначення довжини рядка S типу char* може бути такою:

```
int KolSimv(char* S)
{ int k;                  // Змінна для результату (довжини рядка)
 asm
 { pusha
  mov edi,dword ptr[S]// Завантажити в EDI адресу початка рядка S
  mov esi, edi          // Запам'ятати початок рядка в ESI
  cmp byte ptr[S],0     // Якщо перший символ =0 (рядок пустий),
  je @exit              // перейти на мітку @exit
@nex:
  inc edi               // Перейти на наступний символ рядка
  cmp byte ptr[edi],0 // Допоки не кінець рядка (тобто поки не 0-символ),
  jne @nex              // перейти на мітку, тобто повторювати перевірку,
@exit:
  sub edi, esi          // інакше – обчислити довжину рядка (різницю адрес)
  mov k, edi
  popa
 } return k;            // Функція повертає довжину рядка
}
```

Ця функція при компілюванні займає 29 байтів пам’яті. Порівняно з цим, аналогічна стандартна C++-функція strlen() при компілюванні займає 90 байтів пам’яті. Функція з викликом функції KolSimv() може виглядати, наприклад, як:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char *s = Edit1->Text.c_str();
  int k = KolSimv(s);
  Edit2->Text = IntToStr(k);
}
```

Аналогічна за призначенням функція визначення довжини короткого рядка типу ShortString може мати більш компактний вигляд:

```
int Kol_Short(ShortString S)
{ unsigned char k;
  asm
  { mov al,byte ptr[S]// Завантажити перший байт,
                      // який містить значення довжини рядка,
    mov k, al         // в регістр al та вивантажити його з al до змінної k
  } return k;    }
void __fastcall TForm1::Button2Click(TObject *Sender)
{ ShortString S = Edit1->Text;
  int k = Kol_Short(S);       // Виклик функції  Kol_Short()
  Edit2->Text = IntToStr(k);
}
```

Створимо C++-проект з кількома функцями опрацювання рядків, що містять асемблерні “вставки”. Проект міститиме функції:

– визначення довжини рядка типу AnsiString;

– визначення індексу найменшого елемента (номера символу з найменшим кодом в ASCII-таблиці), починаючи з елемента із заданим індексом і до кінця рядка. У цій функції буде використана попередня функція визначення довжини рядка;

– обміну місцями двох символів із заданими номерами у рядку типу AnsiString;

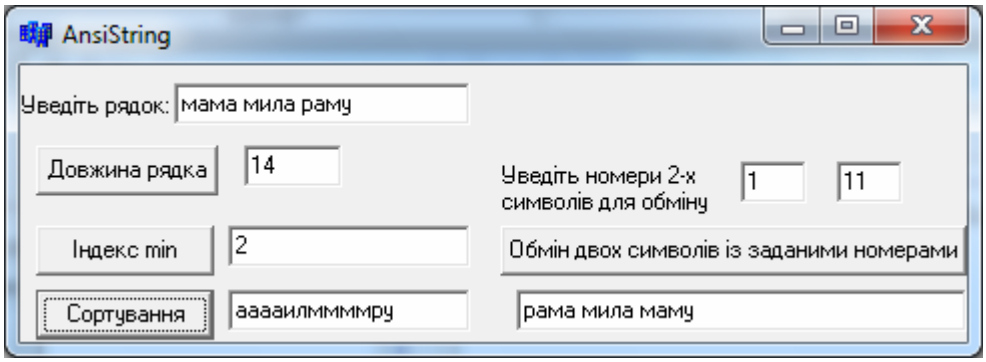
– сортування символів AnsiString-рядка за алфавітом (тобто за зростанням кодів в ASCII-таблиці). У цій функції будуть використані попередні три функції.

У розділі 1.2 зазначалося, що значенням змінної S типу AnsiString, так само як для змінних типу char*, є адреса першого символу відповідного рядка. З цього випливає, що алгоритм визначення довжини AnsiString–рядка може збігатися з розглянутим вище алгоритмом для рядків типу char*. Ви побачите, що представлена нижче функція aLength відрізняється від уже розглянутої функції KolSimv, фактично, лише типом вхідного параметра. Зауважимо також, що наступна функція aLength() може використовуватись для визначення довжини ShortString– та WideString–рядків.

```
int  aLength(AnsiString S) // Функція визначення довжини рядка S
{ int  result;             // Змінна для результату (довжини рядка)
  asm
  { pusha
    mov edi, dword ptr[S]  // Завантажити в edi адресу змінної S
                           // (нагадаємо, що змінна типу AnsiString є вказівником і
                           // містить адресу першого елемента відповідного рядка)
     mov esi, edi          // Запам'ятати в ESI адресу початку рядка
     cmp byte ptr[edi], 0  // Якщо символ за адресою, яка міститься в EDI,
     je @ex                // є нульовим, тобто рядок пустий, перейти на мітку @ex
@nx: inc edi               // Перейти на наступний символ, збільшивши адресу на 1
     cmp byte ptr[edi], 0  // Допоки не кінець рядка, тобто поки не 0-символ,
     jne @nx               // перейти на мітку @nx, тобто повторювати перевірку,
@ex: sub edi, esi          // інакше обчислити довжину рядка (різницю адрес)
     mov result, edi
     popa
    } return result;
}
//-------------------------------------------------------------
// Функція визначення індексу найменшого елемента, починаючи з елемента
// з індексом TailBeg і до кінця рядка S
int aIndMinInTail(AnsiString S, int TailBeg )
{ char min = S[TailBeg]; int j, jmin = TailBeg;
  for(j=TailBeg+1; j<=aLength(S); j++)
    if(S[j]<min )  { min=S[j]; jmin=j; }
  return jmin; }
```

```
//---------------------------------------------------------
// Функція обміну двох символів з індексами i та j
AnsiString aSwap( AnsiString S, int i, int j )
{ asm
  { pusha
    mov edi,dword ptr [S]     // Завантажити адресу початку рядка
    mov esi, edi              // Запам'ятати адресу початку
    cmp byte ptr[edi], 0      // Якщо перший символ за адресою з edi =0
    je @exit                  // (рядок пустий), перейти на @exit
    add edi, i                // В edi – адреса (i+1)-го елемента рядка
    add esi, j
    mov AL, byte ptr [edi-1] // В AL – байт (символ) рядка з номером i
    xchg AL,byte ptr [esi-1] // Обмін місцями AL та символу з номером j
    mov byte ptr [edi-1], AL // На місце номер i записати байт з AL
@exit:
    popa
  } return S;
}
//---------------------------------------------------------
AnsiString aSort(AnsiString S)     // Функція сортування
{ int k, j;
  for(k=1; k<aLength(S); k++)
  { j = aIndMinInTail(S, k);
    aSwap(S, k, j);
  }
  return S;}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject*Sender)//Довжина рядка
{ AnsiString s=Edit1->Text;
  int kol=aLength(s);
  Edit2->Text=IntToStr(kol);
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender) // Індекс min
{ AnsiString s=Edit1->Text;
  int kol=aIndMinInTail(s,1);
  Edit3->Text=IntToStr(kol);}
//---------------------------------------------------------
// Обмін двох символів із заданими номерами
void __fastcall TForm1::Button3Click(TObject *Sender)
{ AnsiString s=Edit1->Text;
  int n1=StrToInt(Edit4->Text),n2=StrToInt(Edit5->Text) ;
  aSwap(s,n1,n2);
  Edit6->Text=s; }
//---------------------------------------------------------
```

```
void __fastcall TForm1::Button4Click(TObject *Sender) // Сортування
{ AnsiString s=Edit1->Text;
  aSort(s);
  Edit7->Text=s;        }
```



2. КОНТРОЛЬНІ ПИТАННЯ

1. Скільки байтів виділяється для змінної типу char? Поясніть, чому не може існувати понад 256 символьних констант. Чому номер 256-ї символьної константи є 255?
2. Запишіть двійковий код символьної константи з десятковим номером 32. Як виглядає такий символ?
3. Запишіть команду асемблера, котра надаватиме символьній змінній `C` значення ‘%’.
4. Запишіть команду асемблера, котра надаватиме символьній змінній `S` значення ‘–’.
5. Запишіть десятковий номер символьної константи з двійковим кодом 0010 0000. Як виглядає такий символ?
6. Як перевірити в асемблері, що символьні змінні `Y` та `Z` зберігають різні символи?
7. Що таке `dword ptr`? Наведіть приклад його використання з поясненням.
8. Запишіть команди асемблера для визначення довжини рядка `S` типу `ShortString`.
9. Чим відрізняються рядки типів `char*` та `AnsiString`?
10. Назвіть ознаку кінця рядка типу `AnsiString`. Наведіть асемблерні команди перевірки символу на цю ознаку.
11. Напишіть фрагмент Assembler-програми, що виконує перевірку символу: чи є він літерою кирилиці.
12. Напишіть фрагмент Assembler-програми, що виконує перевірку символу: чи є він цифрою.
13. Напишіть фрагмент Assembler-програми, що виконує перевірку символу: чи є він великою літерою латиниці.
14. Напишіть фрагмент Assembler-програми, що виконує перевірку символу: чи є він крапкою, чи комою.

15. Назвіть та охарактеризуйте різновиди власних типів рядків у C++ Builder.

16. Чим відрізняються рядки типів `ShortString` та `AnsiString`?

17. Назвіть ознаку кінця рядка типу `char*`. Наведіть асемблерні команди перевірки символу на цю ознаку.

18. Які операції можна виконувати над символами? Наведіть асемблерні команди таких операцій та надайте відповідні пояснення.

19. Як у програмі змінити регістр символів? Наведіть приклади асемблерних команд для перетворення значення символьної змінної `S` з верхнього регістра (великої літери) до нижнього (малої літери).

20. Поясніть призначення асемблерної команди `SCASB`. Наведіть приклад.

21. В який спосіб можна звернутися до окремого символу рядка? Запишіть команду асемблера для надання першому символу рядка значеня “ ” (пробіл).

22. Наведіть відомі способи оголошення рядків C++ Builder та надання їм значення, введеного з компонента `Edit`.

23. Поясніть призначення асемблерних команд `CLD` та `STD`. Наведіть приклад.

24. Як у програмі змінити регістр символів? Наведіть приклади асемблерних команд для перетворення значення символьної змінної `S` з нижнього регістра (малої літери) до верхнього (великої літери).

25. Поясніть, яким є значення змінної `st` та якою є довжина наступного рядка:

```
char *st = "Комп'ютерна програма";
```

Запишіть значення символів `st[0]`, `st[4]`, `st[19]` та `st[20]` цього рядка.

26. Поясніть, яким є значення змінної `st` та якою є довжина наступного рядка:

```
AnsiString st = "Комп'ютерна програма";
```

Запишіть значення символів `st[1]`, `st[5]`, `st[20]` та `st[21]` цього рядка. Чи можна звертатися до символу `st[0]` і чому саме?

27. Поясніть, яким є значення змінної `st` та якою є довжина наступного рядка:

```
ShortString st = "Комп'ютерна програма";
```

Запишіть значення символів `st[0]`, `st[1]`, `st[5]` та `st[20]` цього рядка.

28. Скільки пам'яті буде відведено для рядка, оголошеного як `SmallString<50>`?

29. Поясніть відмінність у оголошеннях змінних `s1` та `s2`. Чи спричинять такі оголошення помилки і чому саме?

```
char s1 = " ", s2 = ' ';
```

30. Поясніть призначення асемблерної команди `rep movsb`. Наведіть приклад.

Лабораторна робота № 4

ОПРАЦЮВАННЯ РЯДКІВ ТА СИМВОЛЬНИХ МАСИВІВ ЗАСОБАМИ ВБУДОВАНОГО В C++ BUILDER АСЕМБЛЕРА

Мета роботи: Вивчення асемблерних команд опрацювання рядків та набуття навичок застосовування їх при створенні проектів в C++ Builder із вбудованим асемблером.

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. РЯДКИ ТИПУ CHAR*

Найбільш поширеним засобом доступу до символів рядка в C++ є вказівники типу `char*`. Так у прикладі

```
char *st = "Комп'ютерна програма";
```

компілятор записує всі символи рядка до масиву і присвоює змінній `st` адресу першого елемента масиву.

Рядок може вважатися порожнім у двох випадках: якщо вказівник на рядок має нульове значення `NULL` (тобто для рядка не відведено пам'ять), або коли вказівник вказує на масив, який складається з одного нульового символу '\0' (тобто рядок не містить жодного значущого символу).

```
char *pc1 = 0;          // pc1 не адресує жодного масиву символів,
const char *pc2 = "";   // pc2 адресує символ '\0'.
```

C++ має багату колекцію функцій опрацювання рядків із завершальним нулем. Якщо через помилку програмування у рядку відсутній нуль-термінатор, функція опрацювання рядка буде продовжувати свою роботу, допоки в пам'яті не зустрінеться `'\0'`. В якості аргументів до функцій переважно передаються вказівники на рядки. Якщо при виконуванні функції здійснюється перенесення символів рядка з місця-джерела до місця-призначення, для рядка-призначення слід завчасно зарезервувати місце в пам'яті. Копіювання рядків з використанням тільки вказівника, без резервування області пам'яті, на яку він вказує – одна з найпоширеніших помилок програмування, навіть у досвідчених програмістів. При виділенні пам'яті для рядка-призначення слід виділяти пам'ять і для нуль-термінатора.

У табл. 5.1 наведено найбільш поширені функції стандартної бібліотеки для роботи з С-рядками. У консолі для використання цих функцій слід залучити до програми бібліотеку `<string.h>`.

Таблиця 5.1. **Деякі функції стандартної бібліотеки <string.h>**

Функція	Призначення	Формат
`strlen()`	Повертає довжину рядка (без урахування символу завершення рядка)	`size_t` **`strlen`**`(char *s);`
`strcat()`	Долучає `s2` до `s1` і, як результат, повертає `s1`	`char *`**`strcat`**`(char *s1, char *s2);`

Закінчення табл. 5.1

Функція	Призначення	Формат
`strcpy()`	Копіює до s1 рядок s2, повертає s1, при цьому попереднє значення s1 втрачається	`char *strcpy (char *s1, char *s2);`
`strncpy()`	Замінює перші `n` символів рядка `s1` на перші `n` символів рядка `s2`	`char *`**`strncpy`** `(char *s1, char *s2, size_t n);`
`strstr()`	Відшукує підрядок s2 в рядку s1, повертає частину рядка s1, розпочинаючи з першого спільного символу для обох рядків і до кінця s1	`char *strstr(char *s1, char *s2);`
`strchr()`	Відшукує перше входження символу `ch` в рядок `s` і повертає вказівник на цей символ, тобто частину рядка `s`, розпочинаючи з символу `ch` і до кінця рядка. Якщо символу `ch` в рядку `s` немає, результат – `NULL`	`char *`**`strchr`**`(char *s, int ch);`
`strrchr()`	Відшукує останнє входження символу в рядку і повертає частину рядка `s`, розпочинаючи з останнього входження символу `ch` і до кінця рядка. Якщо символу `ch` в рядку `s` немає, результат – `NULL`	`char *`**`strrchr`** `(char *s, int c);`
`strspn()`	Повертає довжину початкового сегмента рядка `s1`, символи якого є в рядку `s2`	`size_t` **`strcspn`** `(char *s1, char *s2);`
`strcmp()`	Порівнює рядки і повертає нульове значення, якщо рядки є однакові (`s1=s2`), від'ємне (`s1<s2`) чи додатне (`s1>s2`). Порівняння відбувається посимвольно і в якості результату повертається різниця кодів перших неоднакових символів	`int *`**`strcmp`**`(char *s1, char *s2);`
`strlwr()`	Перетворює всі латинські літери до нижнього регістру	`char *`**`strlwr`**`(char *s);`
`strupr()`	Перетворює всі латинські літери до верхнього регістру	`char *`**`strupr`**`(char *s);`

Розглянемо деякі з наведених у табл. 5.1 функцій детальніше на прикладах, для чого попередньо оголосимо вказівники на рядки `s1`, `s2` та `s3` і надамо їм початкові значення.

```
int k,n; char *s1="На Дерибасівській",*s2=" гарна погода",*s3;
n=strlen(s1);       // Обчислюється довжина рядка s1; n дорівнюватиме 17
k=strlen(s2);       // Обчислюється довжина рядка s2; k=13
k=strcmp(s1,s2);    // k=173; різниця між ASCII-кодами перших неоднакових
                    // символів рядків s1 і s2  дорівнює 173, а оскільки 173>0,
                    // можна стверджувати, що s1>s2
```

```
strcpy(s3,s2);       // s3=" гарна погода" (рядку s3 присвоюється рядок s2)
strncpy(s3,s1,2);    // Копіюються два перші символи з рядка s1 до  s3
s3[2]='\0';          // Третім символом після символів"На" задається
                     // нуль-символ, щоб обрізати рядок
strcat(s1,s2);       // s1="На Дерибасівській гарна погода"; до рядка s1
                     // долучається рядок s2
s3=strchr(s1,' '); // Пошук пробілів у рядку s1;  тепер s3=" Дерибасівській
                     //гарна погода"  – це рядок від першого пробілу до кінця
                     // рядка s1, тобто без першого слова
s3=strrchr(s1,' ');// Пошук останнього пробілу в рядку s1;
                     // тепер s3=" погода"  – це останнє слово рядка s1
n=strchr(s1,' ')-s1+1;// n=3 – індекс першого пробілу в рядку s1, обчислений
                     // як різниця вказівників
k=strcspn(s1," ")+1;// k=3 – індекс першого пробілу в рядку s1; оскільки
                   // нумерація індексів символів розпочинається з 0, слід додати 1
s3=strstr(s1,s2);    // s3 = "гарна погода"; пошук рядка s2 у рядку s1
s3=strupr("C++ Builder"); //  s3="C++ BUILDER" – перетворення усіх
                     // латинських літер рядка до верхнього регістру
s3=strlwr(s3);       // s3="c++ builder" – зворотне перетворення усіх
                     // латинських літер рядка до нижнього регістру
```

Отже, функції `strcpy()` та `strncpy()` призначено для копіювання рядка або його частини до іншого рядка. Функції `strchr()`, `strrchr()` та `strstr()` повертають вказівник на знайдений символ чи підрядок.

При виконанні лабораторного завдання Ви можете самостійно створити аналогічні за дією власні асемблерні функції та порівняти їхні розміри.

1.2. КЛАС РЯДКІВ ANSISTRING (STRING)

Якщо змінна `S` оголошена як `AnsiString`-рядок, більшість функцій опрацювання `S` викликається у формі

```
S.F([параметри])
```

де `F` є іменем відповідної функції.[14] Наприклад, щоб визначити кількість символів `AnsiString`-рядка, використовується функція `Length()` без параметрів; відповідний виклик (з записом кількості символів у змінну `N`) виглядатиме так:

```
int N = S.Length();
```

Деякі найбільш поширені у застосуванні функції та методи опрацювання рядків `AnsiString` наведено у табл. 5.2.

Таблиця 5.2. **Деякі функції та методи опрацювання рядків `AnsiString`**

Назва	Призначення	Формат
`Length()`	Повертає довжину рядка (без урахування символу завершення рядка)	`int` **`Length`**`();`

[14] Така форма виклику зумовлена тим, що `S` є *об'єктом* класу `AnsiString`, а `F` – функцією-*методом* цього класу.

Закінчення табл. 5.2

Назва	Призначення	Формат
SetLength()	Змінює довжину рядка на newLength, за потреби скорочуючи його	AnsiString& **SetLength** (int newLength);
Insert()	Вставляє рядок str, розпочинаючи з індексу index	AnsiString& **Insert**(const AnsiString &str, int index);
Delete()	Вилучає з рядка зазначену кількість символів count, розпочинаючи з індексу index	AnsiString& **Delete** (int index, int count);
Pos()	Повертає номер індексу, з якого розпочинається підрядок subStr. Якщо рядок не містить subStr, функція повертає 0	int **Pos**(const AnsiString &subStr);
SubString()	Повертає підрядок, який містить count символів, розпочинаючи з індексу index	AnsiString **SubString** (int index, int count);
LowerCase()	Перетворює символи рядка на малі, тобто на нижній регістр	AnsiString **LowerCase**();
UpperCase()	Перетворює символи рядка на великі, тобто на верхній регістр	AnsiString **UpperCase**();
c_str()	Перетворює рядок до типу char*	char* **c_str**();

За приклад використання деяких з наведених у таблиці функцій наведемо програмний код

```
AnsiString S = "Рядок символів";
int k=S.Pos(" ");       // Знаходження позиції пробілу; k=6
if(k != 0)
{ S.Insert("чо",k-1);   // Тепер рядок має вигляд "Рядочок символів"
  S.Delete(k+4,4);      // Вилучення 4-х літер з позиції 10;
}                       // тепер рядок має вигляд "Рядочок слів"
S=S.UpperCase();        // S=" РЯДОЧОК СЛІВ"
S=S.SubString(2,1)+S.SubString(9,3);  // S="ЯСЛІ"
S.SetLength(3);         // S=" ЯСЛ"
char *s;
strcpy(s, S.c_str());   // Переведення рядка до типу char*
```

При виконанні лабораторного завдання Ви можете самостійно створити аналогічні за дією власні асемблерні функції та порівняти їхні розміри.

Рядки AnsiString можна переприсвоювати один одному. Також їх можна "склеювати" за допомогою операції конкатенації +:

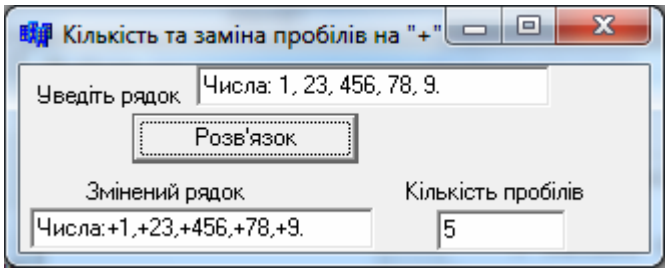
```
AnsiString S1="Hello", S2="world", S3;
S3=S1+" "+S2;       // S3="Hello world"
S3=S1;              // S3="Hello"
```

Рядки AnsiString можна порівнювати за допомогою операцій відношення (==, !=, <, <=, >, >=). Порівняння виконується згідно до ASCII-кодів символів рядка.

1.3. ПРИКЛАДИ ОПРАЦЮВАННЯ РЯДКІВ В АСЕМБЛЕРІ

Приклад 4.1. Організувати асемблерну функцію для визначення кількості пробілів у рядку типу `AnsiString` та заміни їх на символ "+".

```
int kol_pr(AnsiString s, int k)
{ int p;
  asm
  { pusha
 mov edi, dword ptr [s] // Завантажити в EDI адресу 1-го символу рядка s
 mov ecx, dword ptr k   // Завантажити в лічильник довжину рядка у байтах
 mov edx, 0             // Кількість пробілів на початку становить 0
 cmp ecx, 0
 je @exit               // Перевірка на порожній рядок
 cld                    // clear direction flag – після виконання
                        // команди scasb значення EDI ЗРОСТАТИМЕ на 1
 mov al, '  '           // Далі scasb порівнюватиме ES:DI з AL
@next:
 scasb                  // Порівняння AL з рядком і встановлення прапорців
 jne @skip
 inc edx                // Збільшити значення кількості пробілів в EDX на 1 та
 mov byte ptr[edi-1],'+' // замінити цей пробіл у рядку на символ "+"
@skip:
 loop @next          // Якщо ecx>0, перейти до опрацювання наступного символу
@exit:
 mov  dword ptr p, edx  // Завантажити обчислену кількість пробілів до P
 popa
 }
 return p;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S = Edit1->Text;
  int k = S.Length();
  int n = kol_pr(s, k);
   Edit2->Text = s;
   Edit3->Text = IntToStr(n);
}
```

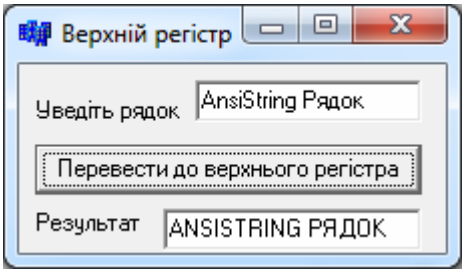


Приклад 4.2. Перевести всі літери рядка до верхнього регістра.

Оскільки різниця поміж великими й малими літерами дорівнює 32, то перетворювання є можливе в такі способи:

– командою віднімання `sub al,32` або `sub al,20h`

– командою встановлення 5-го біту в 0 `and al,11011111b` або `and al,0dfH`

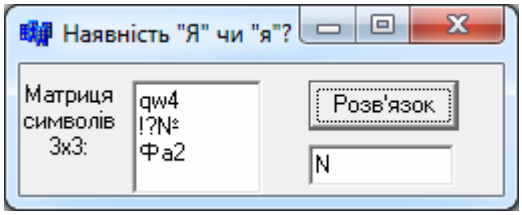


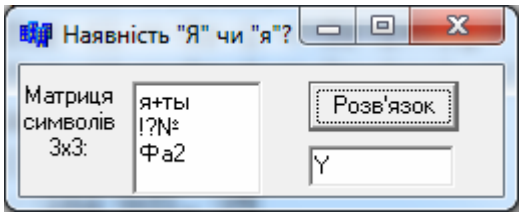

```
void U_case_all(AnsiString s)
{ asm
 { mov edx,dword ptr [s]
 @next:
mov al,[edx]// Завантажити символ до регістра al
cmp al,'a'
jb @no       //  Код  символу є менше за код "a"?
cmp al,'z'
ja @rus      // Якщо символ не є малою латинською літерою, вийти
sub al,32    // інакше перетворити символ у велику латинську літеру командою
             // віднімання sub al,32
jmp @no
 @rus:
cmp al,'a'
jb @no
cmp al,'я'
ja @no       // Якщо символ не є малою літерою кирилиці, вийти інакше перетворити
and al,0dfH // малу літеру кирилиці на велику за рахунок встановлення 5-го біта в 0
 @no:     // командою and al,11011111b або командою sub al,20h або командою sub al,32
mov [edx], al    // Записати перетворену велику літеру на своє місце
lea edx,[edx+1]   // та перейти до наступного символу, збільшивши адресу на 1
cmp al,0          // доки не закінчиться рядок, повторювати перевірку
jne @next
  } return ;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString S = Edit1->Text;
  U_case_all(S);
  Edit2->Text = S;
}
```

Приклад 4.3. Увести символьну матрицю 3x3 й визначити в ній наявність символів ‘Я’ чи ‘я’.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char s[3][3]; unsigned char i, j, k; char c;
 for (i = 0; i < 3; i++)
 for (j = 0; j < 3; j++)
   s[i][j] = Memo1->Lines->Strings[i][j+1];
 asm
 {
   pusha
   lea edi, s      // Завантажити адресу матриці  S
   mov ecx, 9      // Завантажити в лічильник кількість повторів циклу
   mov c, 'N'      // На початку вважатимемо, що символів ‘Я’ чи ‘я’ в рядку нема
```

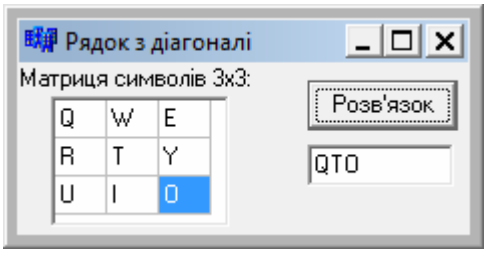
```
@next:
  cmp byte ptr[edi],'я'   // Порівняти символ з 'я'
  jne @NO1          // Якщо символ не збігається з 'я', перейти на мітку @NO1,
  mov  c, 'Y'       // інакше, якщо символ є літерою 'я', змінити змінну C на 'Y'
  jmp  @ex          // та припинити перевірку решти символів
@NO1:
  cmp byte ptr [edi], 'Я'
  jne @NO2
  mov  c, 'Y'
  jmp  @ex
@NO2:
 inc edi
  loop @next
@ex:
  popa
  }
 Edit1->Text = c;
}
```





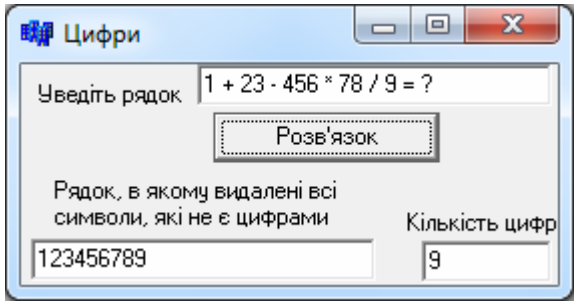
Приклад 4.4. Організувати асемблерну функцію для формування рядка – діагоналі символьної матриці 3x3.

```
AnsiString   fun( char s[3][3])
{ AnsiString rez;    rez.SetLength(3);
 asm
 { pusha
   mov edi, dword ptr [s]     // Завантажити адресу матриці S
   mov esi, dword ptr [rez]   // Завантажити адресу рядка-результату
   mov ecx, 3                 // Завантажити в ECX кількість повторів циклу
 @next:
    mov al,byte ptr [edi]     // Завантажити в AL черговий символ матриці
    mov byte ptr[esi], al     // Вивантажити цей символ до рядка
    lea edi, [edi+4]          // Перейти на наступний елемент діагоналі
    inc esi                   // Перейти на наступний символ рядка
   loop @next
   popa
  }
return rez;
}
```




```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ char s[3][3];
 for(int i=0; i<3; i++)
 for(int j=0; j<3; j++) s[i][j] = StringGrid1->Cells[j][i][1];
 Edit1->Text = fun(s); }
```

Приклад 4.5. Визначити кількість цифр у рядку та видалити з цього рядка всі символи, які не є цифрами. Параметрами наступної функції є рядок s та його фактична довжина k.

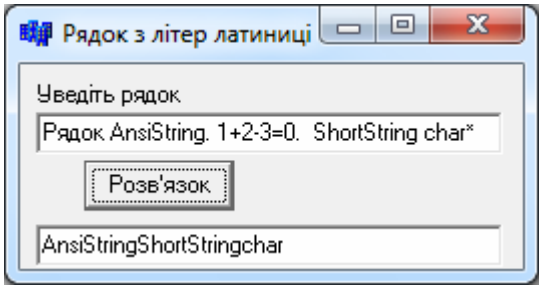



```
int digits(AnsiString s, int k)
{ int kol;
  asm
 { pusha
  mov eax, 0              // Кількість цифр на початку становить 0
  mov edi, dword ptr [s]  // Завантажити  в EDI адресу 1-го символу рядка  s
  mov ecx, dword ptr k    // Завантажити в лічильник довжину рядка у байтах
  cmp ecx, 0
  je @exit                // Перевірка на порожній рядок
@next:
  mov dl,byte ptr [edi]   // Завантажити черговий символ рядка
  cmp dl,'0'              // та перевірити його на інтервал цифр.
  jl @sdvig      // Якщо символ не є цифровий, тобто його код є менший за '0'
  cmp dl,'9'     // чи більший за '9', перейти на мітку @sdvig
  jg  @sdvig     // для видалення цього символу
  inc eax        // Інакше, збільшити значення кількості пробілів в EAX  на 1 та
  jmp @skip      // перейти на мітку, обійшовши видалення нецифрових символів
@sdvig:
  push ecx       // Зберегти значення повторювань зовнішнього циклу (регістр ECX)
  push edi       // та адресу символу (регістр EDI), який перевіряється, оскільки ці
  // регістри використовуватимуться у внутрішньому циклі для видалення нецифрових символів
    @nex:        // З цієї мітки починається внутрішній цикл, в якому видалення
      mov dh,byte ptr [edi+1] // відбувається за рахунок зсуву решти символів
      mov byte ptr [edi],dh   // рядка. При цьому через регістр DH, починаючи
      inc edi    // з нецифрового символу, почергово перевантажуються
      loop @nex  // (зсуваються) всі символи решти рядка
   pop edi       // Після видалення (зсуву) відновлюються значення регістрів EDI та
   pop ecx       // ECX, щоб відновити перевірку, починаючи з номера символу,
   dec edi       // який було видалено, адже він набув нового значення
@skip:
 inc edi         // Перейти на наступний символ, збільшивши адресу в EDI
 loop @next      // Якщо ecx>0, перейти до опрацювання наступного символу
@exit:
 mov kol,eax     // Завантажити обчислену кількість цифр до змінної  kol
 popa
 }  return kol;
}
void __fastcall TForm1::Button1Click(TObject *Sender)
 { AnsiString s = Edit1->Text;
```

```
  int k = digits(s, s.Length());
  Edit2->Text= s;     Edit3->Text= IntToStr(k);  }
```

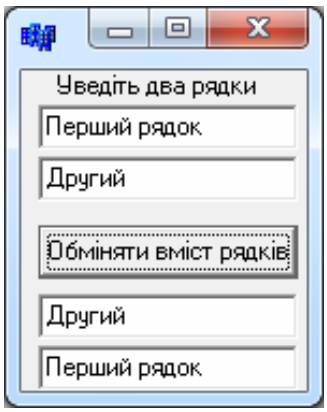
Приклад 4.5. Написати програму, котра вводить рядок символів і будує інший рядок, який міститиме лише літери латиниці з цього рядка.

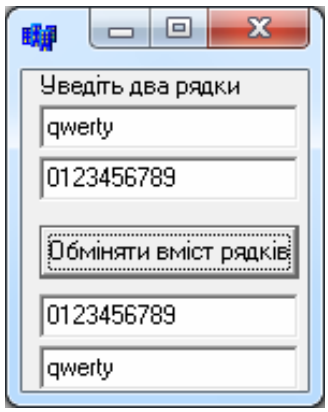
```
AnsiString new_str(AnsiString s)
{ AnsiString rez;  int k= s.Length()  ;
 rez.SetLength(k);        // Встановити довжину рядка по-максимуму
 asm
 { pusha
   mov edi, dword ptr [s]
   mov esi, dword ptr [rez]
   mov ecx, k
   cmp ecx, 0
   je @exit
 @next:
   mov al,byte ptr [edi] // Завантажити черговий символ рядка та
   cmp al,'z'            // перевірити його на інтервали літер латиниці
   jge  @no1
   cmp al,'a'
   jg @yes
 @no1:
   cmp al,'A'
   jl @no2
   cmp al,'Z'
   jg  @no2
 @yes:
   mov byte ptr[esi], al    // Записати вибраний символ у рядок-результат
   inc esi
 @no2:
  inc edi
  loop @next
  mov byte ptr[esi],0      // Позначити кінець рядка-результату нуль-символом
 @exit:
  popa
 }
 return rez;
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString s=Edit1->Text;
  Edit2->Text=new_str(s);
}
```

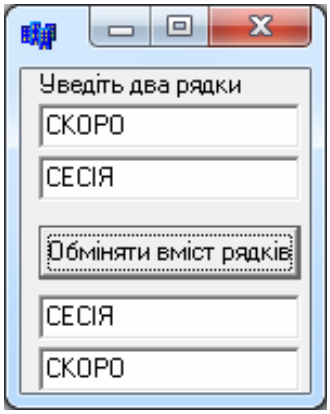


Приклад 4.6. Увести два рядки однакової довжини та обміняти їх вміст, тобто треба обміняти (переслати) їх поміж собою посимвольно (побайтно), використовуючи асемблерні команди MOVSB та REP.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AnsiString s1=Edit1->Text, s2=Edit2->Text, tmp;
 int k1=s1.Length(), k2=s2.Length();
 tmp.SetLength(k1);// Встановити довжину тимчасового рядка
 // Обмін вмісту рядків виконуватимемо в три етапи: 1) tmp ← s1, 2) s1 ← s2, 3) s2 ← tmp.
 asm
 { cld                 // Встановлення прапорця DF = 0, тобто командою MOVSB, яка
  // виконує автоінкремент, рядки опрацьовуватимуться у порядку зростання адрес
  lea esi, s1          // Завантажити адресу першого рядка в ESI
  lea edi, tmp         // Завантажити адресу тимчасового рядка в EDI
  mov ecx, k1          // Кількість повторювань (символів) для команди rep movsb, яка
  rep movsb            // повторює копіювання рядка байтів з [DS:SI] в [ES:DI]
                       // задану в ECX кількість разів, тобто  tmp ← s1
  lea esi, s2          // Наступні чотири команди копіють вміст рядка s2 до s1
  lea edi, s1
  mov ecx, k2
  rep movsb
  lea esi, tmp         // Останній етап:  s2 ← tmp
  lea edi, s2
  mov ecx, k1
  rep movsb
 }
 Edit3->Text=s1;
 Edit4->Text=s2;
}
```







2. ЛАБОРАТОРНЕ ЗАВДАННЯ

1. Розробити програмний проект C++ Builder для введення рядка символів та опрацювання його засобами вбудованого асемблера згідно індивідуального завдання відповідно варіанту.

2. При відлагодженні програми слід використовувати покрокове виконування команд і відстежувати стан регістрів мікропроцесора у вікні дизасемблера.

3. Оформити протокол лабораторної роботи, до якого занести відповіді на відповідні контрольні запитання (вибираються згідно з індивідуальним варіантом на стор. 83) і записати тексти програм.

Варіанти індивідуальних завдань

1. Написати програму, котра вводить рядок символів і знаходить індекс першої коми в цьому рядку.
2. Написати програму, котра вводить рядок символів і замінює в ньому всі коми на пробіли.
3. Написати програму, котра вводить рядок символів і знаходить кількість цифр у цьому рядку.
4. Написати програму, котра вводить рядок символів і замінює всі цифри на пробіли.
5. Написати програму, котра вводить рядок символів і змінює місцями перший та останній введені символи.
6. Написати програму, котра вводить рядок символів і знаходить індекс першої крапки у цьому рядку.
7. Написати програму, котра вводить рядок символів і замінює в ньому всі крапки та коми на знак ‘%’.
8. Написати програму, котра вводить рядок символів і знаходить серед них елемент з найменшим номером у ANSI-таблиці.
9. Написати програму, котра вводить рядок символів і будує інший рядок, який міститиме лише цифри з цього рядка.
10. Написати програму, котра вводить рядок символів і змінює місцями третій та останній введені символи.
11. Написати програму, котра вводить рядок символів і знаходить індекс першого знаку ‘+’ у цьому рядку.
12. Написати програму, котра вводить рядок символів і замінює в ньому всі цифри на знак ‘#’.
13. Написати програму, котра вводить рядок символів і будує інший рядок, який міститиме лише великі латинські літери з першого рядка.
14. Написати програму, котра вводить символьну матрицю розміром 3x4 і визначає, чи містить ця матриця хоча б одну цифру.
15. Написати програму, котра вводить символьну матрицю розміром 3x3 і будує послідовність, котра співпадає з діагоналлю матриці.
16. Написати програму, котра вводить рядок символів і знаходить кількість знаків ‘–’ у цьому рядку.

17. Написати програму, котра вводить рядок символів і будує інший рядок, який міститиме лише великі літери кирилиці з цього рядка.

18. Написати програму, котра вводить символьну послідовність і визначає, чи є в ній символи Ж або ж.

19. Написати програму, котра вводить символьну матрицю розміром 3х4 і будує послідовність, котра співпадає з другим рядком матриці.

20. Написати програму, котра вводить символьну матрицю розміром 3х4 і визначає, чи є у введеній матриці символи Ф та ф.

21. Написати програму, яка вводить символьну матрицю розміром 3х4 і будує послідовність, котра співпадає з четвертим стовпчиком матриці.

22. Написати програму, котра вводить символьну матрицю розміром 3х3 і визначає символ із найбільшим ANSI-номером.

23. Написати програму, котра вводить символьну послідовність і упорядковує елементи введеної послідовності за зростанням кодів ANSI.

24. Написати програму, котра вводить символьну послідовність і замінює в рядку великі літери кирилиці на малі.

25. Написати програму, що вводить символьну матрицю розміром 3х2 і будує послідовність, котра співпадає з першим стовпчиком матриці.

26. Написати програму, котра вводить рядок символів і знаходить кількість літер кирилиці у цьому рядку.

27. Написати програму, котра вводить рядок символів і змінює місцями другий та передостанній введені символи.

28. Написати програму, котра вводить рядок символів і знаходить індекс першої цифри у цьому рядку.

29. Написати програму, котра вводить рядок символів і замінює в ньому всі пробіли на символ ‘+’.

30. Написати програму, котра вводить рядок символів і знаходить кількість символів цього рядка, які *не є* літерами.

КОМПЛЕКСНЕ ЗАВДАННЯ

1. Опрацювання цілих чисел засобами вбудованого в C++ Builder асемблера

Таблиця КЗ.1. **Варіанти індивідуальних завдань**

1	$y=(2-x)^2+(x^2-5x+1)$	**16**	$y=3(x^2-2)^2-12x+1$
2	$y=(x^2+1)(3x-2)-12$	**17**	$y=(4x+1)^2+x(2-x^3)$
3	$y=(3-x^2+7x)-(x+1)^2$	**18**	$y=(2x+3)(x-4)+x^2-1$
4	$y=2(x+1)^2-2x^3+3$	**19**	$y=(x^2+1)^2-2x+3x^3$
5	$y=5x^2-2x+(x-1)^2$	**20**	$y=4(x^2+3x)-(1-x)^2$
6	$y=(1-3x)(2x+5)+x^2$	**21**	$y=3(2x+1)(3-x^2)+2$
7	$y=(1-x)^2+2(x^2+3)$	**22**	$y=(2-x)(x^2+1)+3x+2$
8	$y=2x(x^2+3)-5x+4$	**23**	$y=(3x+7)(1-x^2)+8$
9	$y=(3-x)(x^2+1)+2x^2-1$	**24**	$y=(x+3)(2x-1)^2+3x+4$
10	$y=3x^2+2x-(1+x^2)(11-2x)$	**25**	$y=(x^2+1)(3x-7)-4x$
11	$y=4x^2+3(1+x)(1-2x)$	**26**	$y=(2x+3)^2-(2x+1)(3-x^2)$
12	$y=(2-x^2)^2-(3x+1)(x-4)$	**27**	$y=2x+(x+1)(7-4x)-12$
13	$y=(4x^2-1)(x+5)-2x+3$	**28**	$y=(5-2x)(12+x)-2x^2+1$
14	$y=(2x+1)(x^2-11)-x+7$	**29**	$y=2x^2-1+(2x+5)(x-3)$
15	$y=4x^2+2(x-1)-(3x-5)^2$	**30**	$y=(2x-1)(x+3)-x^2+1$

2. Опрацювання дійсних чисел засобами вбудованого в C++ Builder асемблера

Таблиця КЗ.2. **Варіанти індивідуальних завдань**

1	$Z=\frac{t+y\cos^2 t}{\sqrt{\lvert y+1\rvert}}$	**11**	$P=\frac{\operatorname{ctg}^2 y+h^3}{\sqrt{\lvert y+h\rvert}}$	**21**	$Z=\frac{\sin(p-1)^2}{y^2+\operatorname{tg}(p)}$
2	$D=y^2+\frac{n-1}{\sin^2 y}$	**12**	$U=\frac{\operatorname{tg}(k-\pi)}{2^{k+y}+\sqrt{\lvert y\rvert}}$	**22**	$H=\frac{y^2-y+\sqrt{\lvert y\rvert}}{n^2+\cos n}$
3	$Q=\frac{\sqrt{\lvert k+p\sin k\rvert}}{x-d^3}$	**13**	$D=\frac{a^2+\cos t}{\operatorname{tg}(a-y)+1}$	**23**	$R=\frac{\sqrt{\sin^2 y+k^2}}{\cos(y+k)-y^2}$
4	$F=\sqrt{\lvert y\rvert}+\frac{d^2+1}{\cos(y+\pi)}$	**14**	$F=\frac{\sin(y+1)^2+x}{\cos^2(x-\pi)-\sqrt{\lvert y\rvert}}$	**24**	$W=\frac{\sin v+2^y}{\operatorname{arctg}^2(v/y)}$
5	$R=\frac{\sin(t+\pi)^2+1}{\operatorname{ctg}(t+y)}$	**15**	$T=\frac{\sin(1+u)}{\operatorname{tg}(y+\pi)-u^2}$	**25**	$V=\frac{\left(y+\sqrt{\lvert w\rvert}\right)^3}{\operatorname{tg}(y-\pi)+w^2}$
6	$L=\cos^2 c+\frac{3t^2+4}{\sqrt{c+t}}$	**16**	$L=\frac{y-\cos i}{\operatorname{arctg}\sqrt{\lvert y-i\rvert}+i^3}$	**26**	$E=\frac{\operatorname{ctg}\left(y+q^2\right)}{\sqrt{\lvert y^2+y-1\rvert}+\sin q}$
7	$U=\frac{\cos(k-y)+y^4}{\operatorname{arctg}\sqrt{\lvert y\rvert}-k^2}$	**17**	$W=\frac{t^3+\cos(r)}{2^y\sin^2 r}$	**27**	$T=\frac{\sin\left(h^2-\pi\right)+1}{2^{y+h}+\sqrt{\lvert y\rvert}}$
8	$A=\frac{\sin(y+h)+h^2}{2^h+y}$	**18**	$G=\frac{\cos y-\sin(f+\pi)}{(y+f)^2}$	**28**	$S=\frac{y^3+t\sin(t-\pi)}{\sqrt{\lvert\cos y+t\rvert}}$
9	$R=\frac{\sin^2 y+d}{\operatorname{tg}(d-\pi)^2}$	**19**	$N=\frac{m^2+m-1}{\cos^2 y+\sqrt{\lvert m\rvert}}$	**29**	$K=\frac{t^2+(y-1)^2}{\operatorname{tg}(y)+\cos^2 t}$
10	$G=\frac{w^3-\sqrt{\lvert 1+w\rvert}}{\sin(y+\pi)+\sqrt{\lvert y\rvert}}$	**20**	$T=\frac{t-\sin(t+\pi)}{\sqrt{\lvert y^2-y+1\rvert}}$	**30**	$N=\frac{y^2+\sqrt{y-1}}{\sin(p+y)+2^p}$

3. Організація циклів засобами асемблера

Таблиця КЗ.3. **Варіанти індивідуальних завдань**

1	$\sum_{k=1}^{7}\frac{\sin(x+k)}{(x+1)^2}$	11	$\sum_{k=1}^{12}\frac{\cos(kx)}{k^2+x}$	21	$\sum_{k=1}^{8}\sin^2(x)(k+\cos(x+2))$
2	$\sum_{k=1}^{9}\frac{\mathrm{tg}(x^3-\pi)}{(k+1)^2}$	12	$\sum_{k=1}^{8}\frac{\sin x^k}{4k};\ x=1.75$	22	$\sum_{k=2}^{10}\frac{\mathrm{arctg}^3(2kx)}{(k+x)}$
3	$\sum_{k=1}^{12}\frac{\sin^2(kx)}{x+k}$	13	$\sum_{k=1}^{7}\frac{\mathrm{tg}(x+\pi)^2}{(k-x)}$	23	$\sum_{k=1}^{11}\frac{\sin(x)^3+kx}{k^2}$
4	$\sum_{k=1}^{9}\frac{\mathrm{ctg}(x+1)}{(x+k)^2}$	14	$\sum_{k=1}^{8}k^2\sin\frac{kx^2-\pi}{k}$	24	$\sum_{k=1}^{6}\frac{k^2\sin^2(x/k)}{kx^2+1}$
5	$\sum_{k=1}^{9}\frac{\sin(2kx)+1}{k+x^2}$	15	$\sum_{k=6}^{1}\frac{x^2}{k^3+\cos kx}$	25	$\sum_{k=1}^{12}\frac{\cos(x^2-\pi)}{(x+1)^2+k}$
6	$\sum_{k=1}^{7}\frac{kx\cos(x+k)}{\sin(1-x)+2k}$	16	$\sum_{k=1}^{11}\frac{\mathrm{arctg}(x^2-k)^2}{k^2+1}$	26	$\sum_{k=2}^{9}\frac{\cos^2(k-x)}{\mathrm{tg}(kx)+1}$
7	$\sum_{k=2}^{6}\frac{\sin(k^2-x^2)}{2k+x}$	17	$\sum_{k=1}^{8}\frac{\mathrm{ctg}(x^2-\pi)}{k^2(2x-k)}$	27	$\sum_{k=1}^{7}\frac{k\cos(x+1)^2}{(x+1)^2+k}$
8	$\sum_{k=1}^{8}\frac{\mathrm{tg}(1+kx)}{\mathrm{arctg}(x)+k^2}$	18	$\sum_{k=2}^{9}\frac{\mathrm{tg}(x^2-\pi)}{k^2+1}$	28	$\sum_{k=1}^{10}\cos\left(k^3-\frac{x}{k}\right)^2$
9	$\sum_{k=2}^{9}\frac{\mathrm{tg}(x)-x^2/k}{k^2-1}$	19	$\sum_{k=3}^{10}\frac{x^3\cos x}{k^2+x^2+1}$	29	$\sum_{k=1}^{7}\frac{x\sin(x-k)}{x^2+k}$
10	$\sum_{k=1}^{7}\frac{\mathrm{tg}(k+1)^2}{\sin(x^2-\pi)}$	20	$\sum_{k=3}^{11}\frac{\cos^2(x-\pi)}{2k-1}$	30	$\sum_{k=2}^{6}x\cdot\mathrm{arctg}\frac{x-k}{x+\sin(x-k)}$

ВИМОГИ ТА ПРИКЛАД ВИКОНАННЯ КУРСОВОЇ РОБОТИ

Завдання. Курсова робота виконується в середовищі C++-Builder. Необхідно побудувати окрему бібліотеку (unit), яка міститиме, крім іншого, *не менше трьох реалізованих засобами асемблера підпрограм*, що стосуються відповідно трьох розділів роботи: 1) побудови матриці, 2) побудови вектора, 3) побудови скаляра.

Курсова робота оформлюється на аркушах формату А4 з однієї сторони, з відповідними відступами сторінок для зшивання, та здається у зшитому вигляді чи в спеціальних (пластикових) теках.

Текст курсової роботи повинен містити такі обов'язкові розділи:

1) завдання;

2) текст заголовного файла бібліотеки (`u_AsmLib.h`) з пояснювальними коментарями до кожного оголошення (констант, типів) та кожного прототипу (функції та її параметрів);

3) текст файла реалізації бібліотеки (`u_AsmLib.cpp`) з пояснювальними коментарями:

а) кожного локального імені кожної функції бібліотеки;

б) кожної асемблерної команди з поясненнями:

– які дії вона виконує;

– для чого потрібно виконувати ці дії (яким чином вони сприяють розв'язанню задачі);

– як змінюється вміст регістрів CPU чи FPU внаслідок виконання даної команди;

– які прапорці і як змінюються;

4) текст файла `Unit1.cpp` з пояснювальними коментарями;

5) "контрольні" вхідні дані для кожної функції бібліотеки `u_AsmLib.cpp`, які дозволяли б перевірити правильність її роботи; і результати цих контрольних обчислень;[15]

6) вигляд форми з остаточними результатами обчислень.

Проілюструємо вимоги до курсової роботи на прикладі наступного завдання:

1) обчислити елементи матриці A розміром 3×4 за формулою

$$A_{ij} = \sqrt{i^2 + j^2} - \ln(9 + ij),\ i = 1 \ldots m;\ j = 1 \ldots n;\ m = 3,\ n = 4;$$

2) побудувати вектор X, який є рядком матриці A з мінімальною сумою додатних елементів у рядку;

---

[15] Наприклад, якщо функція обчислює суму додатніх елементів рядків матриці, "контрольна" матриця повинна містити *невеликі додатні* та *від'ємні* елементи, для яких згадані суми легко підрахувати "в умі".

3) для побудованого вектора X обчислити скалярну величину

$$G = \prod_{j=1}^{n} \left(1 + \frac{X_j}{\sum_{k=1}^{j} X_k} \right).$$

Після побудови форми у вікні C++ Builder створимо новий модуль (бібліотеку)[16] з ім'ям `u_AsmLib`, що міститиме, зокрема, такі функції (підпрограми):

1) функцію обчислення елемента матриці A_{ij} за заданими індексами i та j;

2) функцію обчислення всіх елементів матриці A;

3) функцію, яка за заданим номером рядка матриці повертатиме всі елементи цього рядка. Ця функція викликатиметься двічі: перший раз – при обчисленні сум кожного з рядків матриці, а другий – для вибирання елементів вектора X з матриці за визначеним індексом рядка з мінімальною сумою;

4) функцію обчислення за заданим рядком R матриці суми його додатних елементів;

5) функцію обчислення індекса найменшого елемента у заданому векторі сум рядків матриці;

6) функцію обчислення за заданим вектором X скалярної величини G.

Підпрограми 1, 4, 5 та 6 будуть *реалізовані засобами асемблера*.

Приклад виконання курсової роботи

Заголовковий файл бібліотеки `u_AsmLib.h`

```
//---------------------------------------------------------------
#ifndef u_AsmLibH
#define u_AsmLibH

const int m=3, n=4;       // m – кількість рядків, n – кількість стовпчиків матриці
typedef float AType[m][n]; // Оголошення типу матриці
typedef float ARowType[n]; // Оголошення типу рядків матриці
typedef float AColType[m]; // Оголошення типу стовпчика матриці
void AcalcA(AType &A);      // Функція обчислення елементів матриці A
void AGetRow(AType A, int RowNumb, ARowType &R);// Функція за
                            // заданою матрицею A та заданим номером
                            // її рядка RowNumb повертає весь цей рядок – R
float AcalcSum(ARowType R);// Функція обчислює за заданим рядком R
                            // матриці A суму його додатних елементів
int AIndMin(AColType C);   // Функція за заданим вектором типу AColType
                           // повертає індекс найменшого елемента цього вектора
float AcalcProd(ARowType R);// Функція обчислює за заданим рядком R
                             // матриці A величину G
//---------------------------------------------------------------
#endif
```

[16] Командою *File / New / Unit*.

Файл реалізації бібліотеки `u_AsmLib.cpp`

```
#pragma hdrstop
#include "u_AsmLib.h"
#//---------------------------------------------------------------
pragma package(smart_init)

float AcalcAij(int i, int j)  // Функція, яка за заданими i та j обчислює
// елемент Aij матриці A за формулою Aij=sqrt(i*i+j*j)-log(9+i*j); i=1,...,m; j=1,...,n
{ int c9;       // Змінна для зберігання числової константи 9
 float res;     // Змінна для зберігання результату функції AcalcAij
 asm
 { fild i       //  st(0) ← i
   fimul i      //  st(0) ← i * st(0), тобто st(0) ← i^2
   fild j       //  st(0) ← j; st(1) ← i^2
   fimul j      //  st(0) ← j^2; st(1) ← i^2
   fadd         // st(0) ← i^2 + j^2; st(1)=Empty
   fsqrt        // st(0) ← sqrt(i^2+j^2)
   fild i       // st(0) ← i;   st(1) ← sqrt(i^2+j^2)
   fimul j      // st(0) ← i * j;   st(1) ← sqrt(i^2+j^2)
   mov c9,9     // c9 = 9
   fiadd c9     // st(0) ← i j + 9;   st(1) ← sqrt(i^2+j^2)
   fld1         //  st(0) ←  1; st(1) ← i j + 9;   st(2) ← sqrt(i^2+j^2)
   fxch         //  st(0) ← 9+ij;   st(1) ←1;   st(2) = sqrt(i^2+j^2)
   fyl2x        //  st(0) ← st(1)*log2(st0), тобто st(0) ← log2(9+ij); st(1) = sqrt(i^2+j^2)
   fldl2e       //  st(0) ← log2(e);  st(1)=log2(9+ij);  st(2)= sqrt(i^2+j^2)
   fdiv         //  st(0) ← st(1)/st(0), тобто st(0) ← ln(9+ij);  st(1) ← sqrt(i^2+j^2)
   fsub         //  st(0) ← st(1)  – st(0), тобто st(0) ← sqrt(i^2+j^2) – ln(9+ij)
   fstp res     //  res ← st(0);   st(0)=Empty
  }
 return res;
}
//---------------------------------------------------------------
void AcalcA(AType &A)     // Обчислення всіх елементів матриці
{ for(int i=1; i<=m; i++)
  for(int j=1; j<=n; j++)
      A[i-1][j-1] = AcalcAij(i,j);
}
//---------------------------------------------------------------
// Функція повертає рядок R матриці A  за його номером RowNumb
void AGetRow(AType  A, int RowNumb,  ARowType &R)
{ for(int j=0; j<n; j++)
     R[j]=A[RowNumb][j];
}
//---------------------------------------------------------------
```

```
// Функція повертає суму додатних елементів заданого рядка R
float AcalcSum(ARowType R)
{ int k;   ARowType B; // Елементи масиву B тотожні елементам масиву R,
for(k=0;k<n;k++) // але різниця між B та R у тому, що B є ЛОКАЛЬНОЮ величиною
  B[k]=R[k];     // для підпрограми AcalcSum, а R – "зовнішнім" ПАРАМЕТРОМ
                 // цієї підпрограми. У разі параметра, що є масивом, до підпрограми
                 // передається не сам масив R, а ВКАЗІВНИК на нього, тому ЗСУВ
                 // першого елемента відносно адреси, записаної у R,
                 // не є нулем. Отже, вираз R+edx, на відміну від B+edx (див нижче),
                 // не буде правильною адресою поточного елемента масиву
int sf = sizeof(float); // Розмір елемента у байтах – sizeof(float)
 float currEl,  // Поточний елемент
       finSum;  // Остаточна сума
  asm
  { pusha
   mov edx, 0    // Зсув першого елемента відносно початку масиву B
   fldz          // Початкове значення суми дорівнює 0
   mov ecx, n    // Кількість повторювань циклу (елементів рядка)
 @loopBeg:
   mov esi, dword ptr B+edx  // B esi – поточний елемент B_i
   mov currEl, esi       // currEl – змінна для зберігання поточного елемента
   fld currEl            // st(0)=currEl; st(1)= currSum (поточна сума)
   ftst                  // Порівняти st(0) з НУЛЕМ (див. пп. 1.4.5, стор. 46)
                // Якщо st(0)>0, то встановити прапорці C3(ZF)=C2(PF)=C0(CF)=0
   fstsw ax              // Зберегти слово статусу FPU у регістрі AX
   sahf                  // ZF=C3;  PF=C2;  CF=C0 (all=0)
   fstp currEl           // st(0)=currSum (поточна сума); st(1)=Empty
   jbe @newEl            // Пропустити НЕ-додатний елемент
   fadd currEl           // Додати додатний елемент до суми
@newEl:                  // Перейти до наступного елемента, для цього
   add edx, sf           // збільшити зсув на розмір самого елемента sf
   loop @loopBeg         // і, якщо елемент не останній, перейти до @loopBeg
   fstp finSum           // Вивантажити остаточну суму в finSum, st(0)=Empty
   popa
  }
return finSum;
}
//-------------------------------------------------------------
// Функція повертає індекс мінімального елемента заданого стовпця C
int AIndMin(AColType C)
{ AColType B;            // Формування локального для цієї функції вектора B,
  for(int k=0;k<m;k++)B[k]=C[k];// значення елементів якого формуються
                         // з елементів вхідного вектора C
  int indmin;            // Змінна-результат, яка набуде значення індексу min
  int sf=sizeof(float); // Розмір у байтах одного елемента масиву
```

```
  float CurrEl;                  // Поточний елемент стовпчика C
  asm
 { pusha
   mov edx, 0                    // Значення зсуву елементів на початку 0
   mov esi, dword ptr B+edx      // B ESI завантажити поточний елемент
   mov CurrEl, esi               // та скопіювати це значення в змінну  CurrEl
   mov edi, 0                    // B EDI зберігатиметься індекс найменшого
                                 // серед уже розглянутих елементів вектора B
   fld CurrEl            // B st(0) зберігатиметься MIN вектора B, зараз це – B[0]
   mov ecx, m-1                  // Залишилось перевірити m–1  елементів вектора
@2loopBeg:                       // Початок циклу
   add edx, sf                   // Збільшити зсув на розмір одного елемента,
   mov esi, dword ptr B+edx      // щоб перейти до наступного елемента
   mov CurrEl, esi
   fcom CurrEl                   // Порівняти  st(0) з поточним елементом CurrEl
   fstsw ax                      // Зберегти слово статусу FPU  у регістрі AX
   sahf                          // Встановити прапорці ZF=C3; PF=C2;  CF=C0
   jb @2newEl                    // if (st(0)<CurrEl) goto @2newEl, інакше
   ffree st(0)                   // звільнити st(0) від "старого" значення  MIN та
   fld CurrEl                    // записати в st(0) новий поточний мінімум
   mov edi, m                    // Формування в EDI індексу поточного MIN:
   sub edi, ecx                  // EDI = m – (кількість ще не виконаних  кроків)
@2newEl:
   loop @2loopBeg                // Якщо ECX >0, перейти на мітку @2loopBeg
   finit                         // Очистити всі регістри і всі прапорці FPU
   mov indmin, edi               // Зберегти результат
   popa
   }
 return indmin;                  // Повернути результат у точку виклику
}
//-------------------------------------------------------------
```

// Функція повертає значення $G=\prod_{j=1}^{n}\left(1+\frac{X_j}{\sum_{k=1}^{j}X_k}\right)$ для заданого рядка X

```
float AcalcProd(ARowType X)
{ AColType B;              // Формування локального для цієї функції вектора B,
 for(int k=0;k<n;k++)B[k]=X[k];// значення елементів якого формуються
                                 // з елементів вхідного вектора X
  const int sf=sizeof(float);// Розмір у байтах одного елемента вектора
  float CurrEl,                  // Поточний елемент вектора B
  CurrSum,                       // Поточна сума (див. формулу для G)
  CurrProd;                      // Поточний добуток (див. формулу для G)
  asm
```

```
  { pusha
   fld1                          // Початкове значення добутку
   fstp CurrProd                 // заноситься до  CurrProd=1
   fldz                          // Початкове значення суми
   fstp CurrSum                  // заноситься до  CurrSum=0
   mov edx, 0                    // Початковий зсув = 0
   mov esi, dword ptr B+edx      // В esi перший елемент B[0]
   mov ecx, n                    // Кількість співмножників (див. формулу для G)
@3loopBeg:                       // Початок циклу
   mov CurrEl, esi               // Скопіювати поточний елемент до CurrEl,
   fld CurrEl                    // а потім завантажити його в  st(0)
   fld CurrSum                   // st(0) = CurrSum,   st(1) = CurrEl
   fadd st(0), st(1)             // В st(0) нове значення суми CurrSum (див.
                                 // формулу для g); st(1) = CurrEl
   fst CurrSum                   // Зберегти нове значення суми в CurrSum
   fdiv                          // st(0) = CurrEl / CurrSum;  st(1)=Empty
   fld1                          // див. формулу для G
   fadd                          // st(0) = 1 + CurrEl / CurrSum
   fmul CurrProd                 // Домножити CurrProd на st(0)
   fstp CurrProd               // Вивантажити нове значення добутку; st(0)=Empty
   add edx, sf                   // Збільшити зсув на розмір одного елемента,
   mov esi, dword ptr B+edx      // щоб перейти до наступного елемента
   loop @3loopBeg                // Якщо ECX >0, перейти на мітку @3loopBeg
   popa
  }
  return CurrProd;               // Повернути результат у точку виклику
}
```

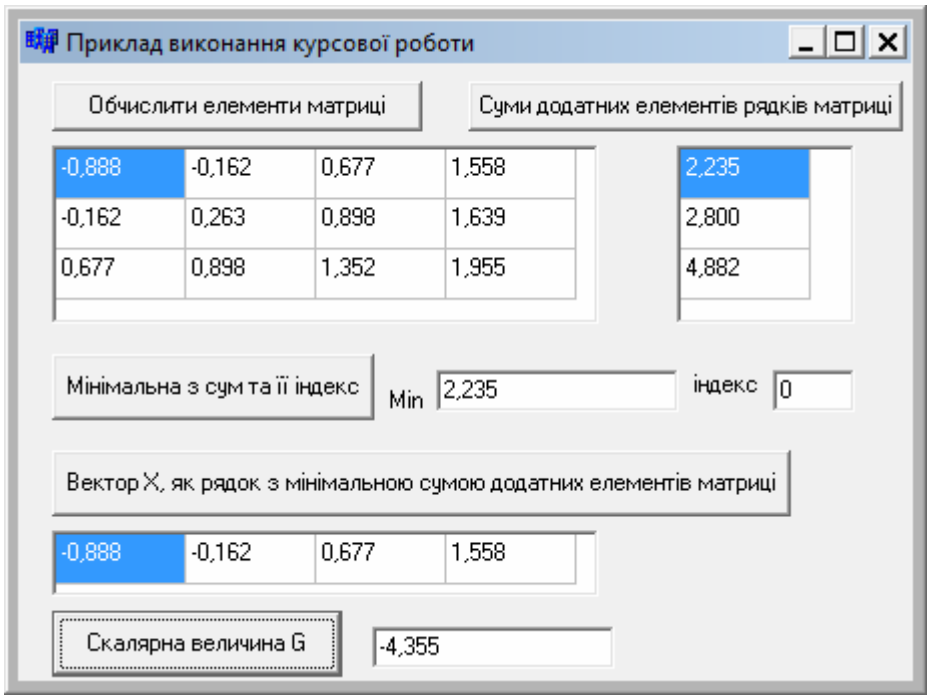
Файл реалізації `Unit1.cpp`

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include "u_AsmLib.h"          //  Долучення власної бібліотеки
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------
```

```
// Обчислити елементи матриці  Aij=sqrt(i*i+j*j)-log(9+i*j); i=1,...,m; j=1,...,n
void __fastcall TForm1::Button1Click(TObject *Sender)
{ AType A;
  AcalcA(A);
  for(int i=0;i<m;i++)
  for(int j=0;j<n;j++)
    StringGrid1->Cells[j][i] = FloatToStrF(A[i][j],ffFixed,7,3);
}
//---------------------------------------------------------------
/* Наведені нижче програми, з метою перевірки їх правильності на РІЗНИХ вхідних даних,
   завжди читають ці вхідні дані  З ВІКОН ФОРМИ. Наприклад, функція Button2Click,
   замість звертання до функції  AcalcA(A), вводить значення елементів матриці A
   з вікна StringGrid1. Отже, якщо відповідні вікна форми не будуть заповнені,
   при натисканні кнопки на формі БУДЕ ВИВЕДЕНО ПОВІДОМЛЕННЯ ПРО ПОМИЛКУ!
*/
// Суми додатних елементів рядків матриці
void __fastcall TForm1::Button2Click(TObject *Sender)
{ AType A;  ARowType R;  int i,j;
  for(i=0;i<m;i++)
  for(j=0;j<n;j++) A[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
  for(i=0;i<m;i++)
  { AGetRow(A, i, R);
    StringGrid2->Cells[0][i] = FormatFloat("0.000", AcalcSum(R));
  }
}
//---------------------------------------------------------------
// Мінімальна з сум та її індекс
void __fastcall TForm1::Button3Click(TObject *Sender)
{ AColType C;
  for(int i=0;i<m;i++) C[i]=StrToFloat(StringGrid2->Cells[0][i]);
  int ind = AIndMin(C);
  Edit1->Text = FloatToStrF(C[ind],ffFixed,7,3);
  Edit2->Text = IntToStr(ind);
}
//---------------------------------------------------------------
// Вектор X, як рядок з мінімальною сумою додатних елементів матриці
void __fastcall TForm1::Button4Click(TObject *Sender)
{ AType A; int i,j,ind;
 for(i=0;i<m;i++)
 for(j=0;j<n;j++) A[i][j]=StrToFloat(StringGrid1->Cells[j][i]);
 ARowType X;
 ind = StrToInt(Edit2->Text);
 AGetRow(A, ind, X);
 for(j=0;j<n;j++)
   StringGrid3->Cells[j][0]=FormatFloat("0.000", X[j]);
}
```

```
//---------------------------------------------------------------------
// Скалярна величина G
void __fastcall TForm1::Button5Click(TObject *Sender)
{ ARowType X;
  for(int j=0;j<n;j++) X[j]=StrToFloat(StringGrid3->Cells[j][0]);
  float G = AcalcProd(X);
  Edit3->Text = FormatFloat("0.000", G);
}
```

Форма з остаточними результатами обчислень



ВАРІАНТИ КУРСОВИХ ЗАВДАНЬ

Кожний варіант завдання містить:

1) формулу для обчислення елементів a_{ij} матриці A;
2) правило для обчислення елементів вектора $X = <x_0, x_1, \ldots>$;
3) правило для обчислення скалярної величини $G(x_0, x_1, \ldots)$.

Таблиця КР.1. **Варіанти індивідуальних завдань**

№ вар.	Індивідуальне завдання
1	1 $a_{ij}=\sin\left(\frac{1}{i+j+1}\right)+2\cos^2\left(\frac{j+i}{4}\right)$, де $i=1,...,n;\ j=1,...,n;\quad n=4.$ 2 Вектор X – сума першого та п'ятого стовпчиків матриці. 3 $G=\min\limits_{1\le i\le n}\{x_i\cdot x_{n-i}\}$, де $n=4.$
2	1 $a_{ij}=3^j i-\lvert 3-i+j\rvert\frac{2+j}{i+2(j-1)}$, де $i=1,...,m;\ j=1,...,n;\ m=3;n=4.$ 2 x_i – сума додатних елементів i - го рядка матриці. 3 $G=\max\limits_{1\le i\le m}\{\lvert x_i-1\rvert\}$, де $m=3.$
3	1 $a_{ij}=\left(\sqrt[3]{\lvert i+j\rvert}\right)/(i+1)-(4+j)$, де $i=1,...,m;\ j=1,...,n;\ m=4;\ n=3.$ 2 x_i – середнє арифметичне модулів елементів i - го рядка. 3 $G=\min\limits_{1\le i\le m}\{x_i^2-x_i\}$, де $m=4.$
4	1 $a_{ij}=2^j(i-4.3)\left(\lvert 3.7-j\rvert-2\right)$, де $i=1,...,m;\ j=1,...,n;\ m=3;\ n=4.$ 2 Вектор X – той зі стовпчиків матриці, другий елемент якого є максимальним елементом другого рядка матриці 3 $G=\sum\limits_{i=1}^{m}x_i^2\cdot\sum\limits_{k=1}^{i}x_k^3.$
5	1 $a_{ij}=\frac{8-j}{i+1}\sqrt{(i+1)^3}+j^2$, де $i=0,...,2;\ j=0,...,4.$ 2 Вектор X – рядок матриці A з найбільшою сумою елементів. 3 $G=\prod\limits_{i=0}^{4}\frac{x_i}{\sum\limits_{k=0}^{i}x_k}$
6	1 $a_{ij}=(i+1)^2\left(2^i(j-3.5)+1.2\right)\left(2-(j+0.5)^2\right)$, де $i=1,...,m;\ j=1,...,n;m=3;n=4.$ 2 Вектор X – рядок матриці A з найменшою сумою елементів. 3 $G=\sum\limits_{i=1}^{n}x_i\Big/\sum\limits_{i=1}^{n}\lvert x_i\rvert.$

Продовження табл. КР.1

№ вар.	Індивідуальне завдання
7	1 $a_{ij}=4^{i+1}3^{i+j}\cos\left(\frac{(i+2)\pi}{6}\right)$, де $i=1,...,m;\ j=1,...,n;\ m=3;\ n=5.$ 2 Вектор X – рядок матриці A, в якому сума першого та другого елементів є максимальною. 3 $G=\cos\left(\prod_{i=1}^{n}x_i^2\right)+\sin\left(\sum_{i=1}^{n}x_i\right).$
8	1 $a_{ij}=3^{j}\left(\frac{(i+2)+j}{2}\right)\ln\frac{i+1}{j+1}$, де $i=1,...,m;\ j=1,...,n;\ m=3;n=4.$ 2 x_i – середнє арифметичне елементів i-го стовпчика. 3 $G=\prod_{i=1}^{n}x_i\sqrt[3]{\sum_{k=1}^{i}x_k^2}.$
9	1 $a_{ij}=3^{i+j}\sqrt{(i+2.6)+j}\,\frac{2\lvert i-j+3\rvert}{3(i+1.4)}$, де $i=1,...,n;\ j=1,...,n;\ n=5.$ 2 X – сума третього та четвертого рядка матриці A. 3 $G=\sum_{i=1}^{n-1}\left(\frac{1}{1+x_i}+x_{i+1}\right).$
10	1 $a_{ij}=2^{j}(i-3.4)\left(\frac{j}{2}+1\right)$, де $i=1,...,m;\ j=1,...,n;\ m=3;\ n=4.$ 2 X – стовпчик матриці A з найбільшою вагою $W_j=\sum_{i=1}^{m}\lvert a_{ij}\rvert.$ 3 $G=\left(1+\sum_{i=1}^{m}x_i\cdot\sum_{k=1}^{i}x_k\right).$
11	1 $a_{ij}=2i^2\frac{3+i}{4+j}+4\frac{i+1.5}{j+1}$, де $i=1,...,m;\ j=1,...,n;\ m=5;\ n=3.$ 2 X – вектор добутків елементів рядків матриці A. 3 $G=\sqrt[3]{\left\lvert\sum_{i=1}^{m}x_i^2\cdot\prod_{k=1}^{i}x_k\right\rvert}.$
12	1 $a_{ij}=2^{j+1}\left(\lvert j-3\rvert+1.3\right)\left(2\lvert 3.3-i\rvert\right)(6.5-j)$, де $i=1,...,n;\ j=1,...,n;\ n=4.$ 2 X – вектор з найменших елементів стовпчиків матриці A. 3 $G=\sqrt[3]{\sum_{i=1}^{n}x_i^2\prod_{k=1}^{i}x_k}$

Продовження табл. КР.1

№ вар.	Індивідуальне завдання
13	1 $a_{ij}=\sqrt[3]{i+(j+1.4)^2}-\lvert i+j-5\rvert$, де $i=1,...,m$; $j=1,...,n$; $m=3$; $n=5$. 2 X – рядок матриці A, перший елемент якого є максимальним елементом її першого стовпчика. 3 $G=\sum\limits_{i=1}^{n}\left(x_i+\prod\limits_{k=1}^{i}x_k\right)$.
14	1 $a_{ij}=2^{i+j}\left(i+4-j+\cos(ij)\right)$, де $i=1,...,n$; $j=1,...,n$; $n=4$. 2 x_i – скалярний добуток i - го стовпчика матриці A на її 2 - й рядок. 3 $G=4\ln\left\lvert\sum\limits_{i=1}^{n}\dfrac{x_i+2}{\prod\limits_{k=1}^{i}(1+x_k)}\right\rvert$.
15	1 $a_{ij}=i^3\times\sqrt{j+1}-\dfrac{2+j}{\lvert 2-j\rvert+1}$, де $i=1,...,n$; $j=1,...,n$; $n=5$. 2 x_i – скалярний добуток i - го рядка матриці A на її третій стовпчик. 3 $G=\sum\limits_{i=1}^{n}\dfrac{1+\prod\limits_{k=1}^{i}x_k}{1+\lvert x_i\rvert}$.
16	1 $a_{ij}=2^{i+1}(i-3.9)\left(j\left\lvert 2-\dfrac{3}{i+1}\right\rvert\right)$, де $i=1,...,m$; $j=1,...,n$; $m=4$; $n=5$. 2 x_i – найбільший елемент i - го рядка матриці A. 3 $G=\sqrt{\dfrac{\sum\limits_{i=1}^{m}x_i^2}{\left\lvert\prod\limits_{i=1}^{m}x_i\right\rvert}}$.
17	1 $a_{ij}=3.5^{2i}\lvert i-3j\rvert+\dfrac{2+j}{i+j}$, де $i=1,...,m$; $j=1,...,n$; $m=3$; $n=6$. 2 x_i – сума тих елементів i-го рядка матриці A, які мають парний другий індекс. 3 $G=\sum\limits_{i=1}^{m}\left(\dfrac{\prod\limits_{k=1}^{i}(1+x_k)}{1+x_i}\right)^2$.

Продовження табл. КР.1

№ вар.	Індивідуальне завдання
18	1 $a_{ij} = 4^{2+i}\left(i - \dfrac{i+1}{4+i}\right)\sin(4ij+1.5))$, де $i = 1,...,m;\ \ j = 1,...,n;\ \ m = 5;\ \ n = 4.$ 2 x_j – добуток тих елементів j-го стовпчика матриці A, які мають непарний перший індекс. 3 $G = \sum\limits_{i=1}^{n} x_i \left(1 + \prod\limits_{k=1}^{i} x_k\right).$
19	1 $a_{ij} = \dfrac{3^i + ij + 0.5}{4 + i + j}\sin(i + j),$ де $i = 1,...,n;\ \ j = 1,...,n;\ \ n = 5.$ 2 X – бічна діагональ матриці A. 3 $G = \ln\left(\prod\limits_{i=1}^{n}(1 + \lvert x_i\rvert)\right) - \ln\left(\sum\limits_{i=1}^{n}\lvert x_i\rvert\right).$
20	1 $a_{ij} = \dfrac{i + (j+1.7)^2}{2 + i + j} 3^i e^{2(i+2)},$ де $i = 1,...,n;\ \ j = 1,...,n;\ \ n = 4.$ 2 X – вектор - сума першого стовпчика і головної діагоналі матриці A. 3 $G = \prod\limits_{i=1}^{n}\left(1 + \dfrac{3x_i}{\sum\limits_{k=1}^{i} x_k}\right).$
21	1 $a_{ij} = \sqrt{(i+1.2)^2 + j}\sin(i + 2 + j) + 2.3,$ де $i = 1,...,m;\ \ j = 1,...,n;\ \ m = 3;\ \ n = 5.$ 2 X – вектор добутків елементів стовпчиків матриці A. 3 $G = \sum\limits_{i=1}^{m}\left(x_i^2 - \prod\limits_{k=1}^{i}(\lvert x_k\rvert + 1)\right).$
22	1 $a_{ij} = 3^{i+j}(5 + i)\sin(i(j + 2.3))$, де $i = 1,...,m;\ \ j = 1,...,n;\ \ m = 4;\ \ n = 3.$ 2 X – стовпчик матриці A з найменшою сумою елементів. 3 $G = \dfrac{\prod\limits_{i=1}^{m}(1 + \ln\lvert x_i\rvert)}{\left(\sum\limits_{i=1}^{m} x_i\right)}.$

Продовження табл. КР.1

№ вар.	Індивідуальне завдання
23	1 $a_{ij} = \sqrt[3]{(i+1)+j} \times \sin\left(\frac{i+12}{2}\right)$, де $i = 1,...,n;\ j = 1,...,n;\quad n = 5.$ 2 X – стовпчик матриці A, для якого сума першого та останнього елемента є мінімальною порівняно з іншими стовпчиками. 3 $G = \sum_{i=1}^{n} \left(\ln\lvert x_i\rvert \cdot \ln\lvert x_{n+1-i}\rvert\right).$
24	1 $a_{ij} = \sqrt{(i+2)^2 + j}\,(i+1)^2 \sin(i+2j)$, де $i = 1,...,m;\ j = 1,...,n; m = 4; n = 3.$ 2 x_i – сума від'ємних елементів i-го рядка матриці A. 3 $G = \sum_{i=1}^{m} \left(x_i^2 - \prod_{k=1}^{i} \left(\lvert x_k\rvert + 1\right)\right).$
25	1 $a_{ij} = \ln\left(1 + \lvert i - j\rvert + e^{\sqrt{i+j}}\right)$, де $i = 1,...,n;\ j = 1,...,n;\quad n = 4.$ 2 X – вектор - сума головної діагоналі та 3 - го рядка матриці A. 3 G дорівнює індексу найменшого елемента вектора X.
26	1 $a_{ij} = \sqrt{(i+1.6)^2 + j^2} - 2.3\ln(3+ij)$, де $i = 1,...,n;\ j = 1,...,n;\quad n = 4.$ 2 x_i – максимальний від'ємний елемент i - го рядка матриці A. 3 $G = \prod_{i=1}^{n} \left(1 + \frac{x_i}{\sum_{k=1}^{i} x_k}\right).$
27	1 $a_{ij} = 2 + i^2\left(\sqrt{i+1} + \sqrt{j+1}\right) - 1.8\mathrm{e}^{i}$, де $i = 1,...,m;\ j = 1,...,n; m = 3; n = 4.$ 2 x_j – мінімальний додатний елемент j - го стовпчика матриці A. 3 $G = \sum_{i=1}^{m} \frac{x_i^2 + 2x_i}{\prod_{k=1}^{i} (1 + x_k)}.$
28	1 $a_{ij} = 3^{i+j}\sqrt[3]{i+j+2j}\sin(ij)$, де $i = 1,...,n;\ j = 1,...,n;\quad n = 5.$ 2 X – стовпчик матриці A з найбільшою сумою квадратів його елементів. 3 $G = \frac{\prod_{i=1}^{n} \left(1 + \ln\lvert x_i\rvert\right)}{\left(\sum_{i=1}^{n} i x_i\right) + 2.5}.$

Закінчення табл. КР.1

№ вар.	Індивідуальне завдання
29	1 $a_{ij} = \left(\lvert i-4\rvert + 2\lvert 3{,}5 - j\rvert\right)\left(6 - 0{,}7\, j^2\right)$, де $i = 1,\ldots,m;\ j = 1,\ldots,n; m = 4; n = 3.$ 2 X – рядок матриці A, найменш віддалений від першого рядка. Відстань між першим та i - тим рядком обчислюється за формулою: $$r_i = \sum_{j=1}^{n} \lvert a_{0j} - a_{ij}\rvert.$$ 3 $G = \dfrac{\text{добуток елементів вектора } x}{\text{сума елементів вектора } x}.$
30	1 $a_{ij} = (-1.5)^{i-j}\left(\dfrac{i + (j + 2.1)}{3.5} - 1.1\right)$, де $i = 0,\ldots,m;\ j = 0,\ldots,n;\ m = 3;\ n = 5.$ 2 X – стовпчик матриці A, найбільш віддалений від останнього стовпчика. Відстань між j - тим та останнім стовпчиком обчислюється за формулою: $$d_j = \frac{1}{m} \cdot \sqrt{\sum_{i=1}^{m} \left(a_{ij} - a_{in}\right)^2}.$$ 3 $G = \prod_{i=1}^{m} x_i \sum_{k=1}^{i} x_k.$

СПИСОК ВИКОРИСТАНОЇ ТА РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1) Абель П. Язык ассемблера для IBM PC и программирования [Текст] / П. Абель; пер. с англ. – М. : Высшая школа, 1992. – 447с.
2) Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT, AT [Текст] / Роберт Джордейн. – М. : Финансы и статистика, 1992. – 544 с.
3) Зубков С. В. Assembler для DOS, Windows и Unix [Текст] / С. В. Зубков. – М. : ДМК Пресс; СПб. : Питер, 2004. – 608 с.
4) Магда Ю. С. Ассемблер для процессоров Intel Pentium [Текст] / Ю. С. Магда. – СПб.: Питер, 2006. – 410 с.
5) Магда Ю. С. Ассемблер. Разработка и оптимизация Windows-приложе-ний [Текст] / Ю. С. Магда. – БХВ-Петербург, 2003. – 544 с.
6) Майко Г. В. Assembler для IBM PC [Текст] / Г. В. Майко. – М.: Бизнес-информ, 1999. – 212 с.
7) Сван Т. Освоение Turbo Assembler [Текст] / Том Сван; пер. с англ. – К. – М. – СПб. : Диалектика, 1996. – 544 с.
8) Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблер [Текст] / Л. Скэнлон. – М. : Радио и связь, 1989. – 336 с.
9) Юров В. Assembler: учебный курс [Текст] / В. Юров, С. Хорошенко. – СПб.: Питер Ком, 1999. – 672 с.

ЗМІСТ

ПЕРЕДМОВА 3
ВСТУП 5

Тема № 1. **АПАРАТНІ ЗАСОБИ КОМП'ЮТЕРІВ** 6
1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 6
1.1. Процесор IBM PC-сумісного комп'ютера 6
1.2. Принципи адресації IBM PC 11
1.3. Переривання 15
1.4. Порти 17
2. КОНТРОЛЬНІ ПИТАННЯ 18

Тема № 2. **ОСНОВНІ КОМАНДИ АСЕМБЛЕРА ДЛЯ ОПРАЦЮВАННЯ ЦІЛИХ ЧИСЕЛ** 20
1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 20
1.1. Формат команд 20
1.2. Форми подання чисел 21
1.3. Змінні 21
1.4. Оператори для записування виразів у операндах 22
1.5. Команди пересилання даних 22
1.6. Арифметичні команди 24
1.7. Команди зсуву 27
2. КОНТРОЛЬНІ ПИТАННЯ 28

Лабораторна робота № 1
АРИФМЕТИЧНІ ОПЕРАЦІЇ НАД ЦІЛИМИ ЧИСЛАМИ У ВБУДОВАНОМУ В C++ BUILDER АСЕМБЛЕРІ 31
1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 31
1.1. Початкові відомості про вбудований У C++ Builder асемблер 31
1.2. Покрокове виконування програми та застосування дизасемблера при налагодженні програмного коду 31
1.3. Приклади програм із вбудованим асемблером для виконання арифметичних операцій 33
2. ЛАБОРАТОРНЕ ЗАВДАННЯ 35

Тема № 3. **РОБОТА В АСЕМБЛЕРІ З ДІЙСНИМИ ЧИСЛАМИ** 37
1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 37
1.1. Співпроцесорні конфігурації 37
1.2. Регістри FPU 37
1.3. Числа FPU 38
1.4. Деякі команди FPU 39
2. КОНТРОЛЬНІ ПИТАННЯ 45

Лабораторна робота № 2

АРИФМЕТИЧНІ ОПЕРАЦІЇ НАД ДІЙСНИМИ ЧИСЛАМИ У ВБУДОВАНОМУ В C++ BUILDER АСЕМБЛЕРІ 47

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 47
   1.1. Приклади використовування команд FPU у вбудованому асемблері 47
   1.2. Перегляд стану регістрів CPU під час покрокового виконання програми 54
2 ЛАБОРАТОРНЕ ЗАВДАННЯ 54

Тема № 4. **РОЗГАЛУЖЕННЯ ТА ЦИКЛИ В АСЕМБЛЕРІ** 56

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 56
   1.1. Логічні команди асемблера 56
   1.2. прапорці регістрів flags та Eflags 57
   1.3. Деякі асемблерні команди передачі керування 60
   1.4. Приклади організації розгалужень та циклів в асемблері 62
2. КОНТРОЛЬНІ ПИТАННЯ 64

Лабораторна робота № 3

ОПРАЦЮВАННЯ ЧИСЛОВИХ МАСИВІВ ЗАСОБАМИ ВБУДОВАНОГО В C++ BUILDER АСЕМБЛЕРА 66

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 66
   1.1. Засоби асемблера для опрацювання масивів 66
   1.2. Приклади опрацювання одновимірних масивів в асемблері 68
2. ЛАБОРАТОРНЕ ЗАВДАННЯ 73

Тема № 5. **ОПРАЦЮВАННЯ РЯДКІВ ЗАСОБАМИ АСЕМБЛЕРА** 75

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 75
   1.1. Деякі асемблерні команди роботи з рядками та блоками даних 75
   1.2. Типи рядків У C++ Builder 76
   1.3. Організація асемблерних функцій 79
2. КОНТРОЛЬНІ ПИТАННЯ 83

Лабораторна робота № 4

ОПРАЦЮВАННЯ РЯДКІВ ТА СИМВОЛЬНИХ МАСИВІВ ЗАСОБАМИ ВБУДОВАНОГО В C++ BUILDER АСЕМБЛЕРА 85

1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ 85
   1.1. Рядки типу char* 85
   1.2. Клас рядків AnsiString (String) 87
   1.3. Приклади опрацювання рядків в асемблері 89
2. ЛАБОРАТОРНЕ ЗАВДАННЯ 95

КОМПЛЕКСНЕ ЗАВДАННЯ 97

ВИМОГИ ТА ПРИКЛАД ВИКОНАННЯ КУРСОВОЇ РОБОТИ 100

ВАРІАНТИ КУРСОВИХ ЗАВДАНЬ 107

СПИСОК ВИКОРИСТАНОЇ ТА РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 114