МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЖИТОМИРСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

АРХІТЕКТУРА КОМП'ЮТЕРА

Частина 2

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДЛЯ ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Житомир
2018

УДК 004.23
А-63

Рекомендовано до друку науково-методичною радою
Житомирського державного технологічного університету
(протокол № 2 від 17 жовтня 2018 року)

Рецензенти:
І. В. Пулеко – кандидат технічних наук, доцент
І. І. Сугоняк – кандидат технічних наук, доцент

А-63 Архітектура комп'ютера : методичні рекомендації для виконання лабораторних робіт. Ч. 2 / підг. А. А. Єфіменко Є. М. Байлюк, О. А. Покотило. – Житомир : ЖДТУ, 2018. – 58 с.

Методичні рекомендації містять теоретичний матеріал, приклади та вказівки для виконання лабораторних робіт, пов’язаних із вивченням питань представлення даних в пам’яті комп’ютера.

Методичні рекомендації призначені для студентів, які навчаються за спеціальностями 121 „Інженерія програмного забезпечення”, 123 „Комп’ютерна інженерія”, 125 „Кібербезпека” галузі знань 12 „Інформаційні технології”.

Підготували: кандидат технічних наук **А. А. Єфіменко, Є. М. Байлюк, О. А. Покотило**.

УДК 004.23

ЗМІСТ

ВСТУП

Дані методичні рекомендації розроблені для студентів, які навчаються за спеціальностями 121 „Інженерія програмного забезпечення", 123 „Комп'ютерна інженерія", 125 „Кібербезпека" галузі знань 12 „Інформаційні технології", для вивчення змістовного модуля „Програмування на мові Ассемблер. Обчислення математичних задач" з дисципліни „Архітектура комп'ютера". Зазначена дисципліна згідно з освітньою програмою та навчальними планами підготовки бакалаврів належить до нормативної частини циклу дисциплін професійної і практичної підготовки.

Основним призначенням даного видання є: поглиблення теоретичних знань, отриманих студентами під час вивчення змістового модуля; набуття практичних навичок з написання програм на мові Ассемблера для обчислення математичних задач.

Слід зазначити, що роботи виконуються індивідуально за варіантами. Вибір номера варіанта проводиться за списком групи.

Повне виконання лабораторної роботи передбачає виконання таких етапів: ознайомлення з теоретичними відомостями; аналіз прикладу розрахунків; аналіз індивідуального варіанта завдання; формування висновків по роботі; оформлення та захист звіту; підготовка до подальших контрольних заходів.

Звіт із лабораторної роботи оформлюється згідно з вимогами Державного стандарту України ДСТУ 3008-95 „Документація. Звіти у сфері науки і техніки. Структура і правила оформлення", міжнародних стандартів ISO 5966:1982, ГОСТ 19.404 ЕСПД „Пояснительная записка. Требования к содержанию и оформлению" і повинен містити такі складові: номер роботи; тема роботи; мета роботи; розділ, у якому описано хід виконання роботи відповідно до пунктів завдання; висновки; додаток із лістингами програм, рукописні відповіді на контрольні питання. Звіт виконується в електронному вигляді, роздруковується, доповнюється відповідями на контрольні питання і здається на кафедру для перевірки та захисту.

Під час оформлення звіту необхідно дотримуватися таких вимог: звіт оформлюється на аркушах формату А4 з рамками (перша сторінка – з кутовим штампом форми 2, решта сторінок – форми 2а за ГОСТ 2.104-68 ЕСКД. „Основные надписи"). Штампи заповнюються згідно з вимогами. Нумерація сторінок наскрізна в межах ро-

боти. Поля для тексту: ліве – 25 мм, праве – 10 мм, верхнє – 20 мм від краю аркуша, нижнє – 10 мм від верхнього краю штампа. Текст роботи оформлюється шрифтом Times New Roman 14-го розміру, міжрядковий інтервал 1 – 1,5. Абзацний відступ 5 символів. Ілюстрації та таблиці повинні розміщуватися безпосередньо після тексту, де вони зустрічаються.

Ілюстрації (креслення, рисунки, схеми тощо) позначаються таким чином: „Рисунок № – Назва рисунка" (вирівнювання по центру, без абзацного відступу). Позначення ілюстрації виконується під ілюстрацією. Якщо кількість ілюстрацій значна і вони однотипні та невеликі за розміром, то їх рекомендується групувати в одну ілюстрацію, позначати літерами *а*), *б*) ... і підписувати їх як одну ілюстрацію.

Таблиці позначаються таким чином: „Таблиця № – Назва таблиці" (вирівнювання по лівому краю, без абзацного відступу). Їх нумерація здійснюється окремо і наскрізно у межах роботи. Позначення таблиці виконується над таблицею. Текст таблиць рекомендується оформлювати шрифтом New Roman 12-го розміру, міжрядковий інтервал 1. Якщо таблиця займає понад одну сторінку, то на другій і наступних сторінках ставиться позначення „Продовження табл. №". Головка таблиці повторюється в усіх її частинах.

У тексті звіту можна використовувати переліки (списки в термінології текстових редакторів). Перед перерахуванням ставиться двокрапка. Перед кожною позицією переліку можна ставити тире, малу літеру українського алвафіту з дужкою, арабські цифри з дужкою. Елементи переліку пишуться з маленької літери, наприкінці ставиться крапка з комою, для останнього елемента – крапка.

Важливо, щоб контрольні питання, які наводяться після кожної лабораторної роботи, були ретельно відпрацьованими студентом самостійно з метою підготовки до контрольних заходів, таких як тестування та вирішення практичних завдань.

Лабораторна робота № 4
ОБЧИСЛЕННЯ ПРОСТИХ ЦІЛОЧИСЛЕНИХ ВИРАЗІВ НА МОВІ ASSEMBLER

Мета заняття: ознайомитися з типами цілочисельних даних платформ Win16 та Win32; ознайомитися з основними командами мови Assembler; набути практичних навичок в написанні програм для обчислення простих цілочисельних виразів на мові Assembler.

Теоретичні відомості

Типи цілочисельних даних платформ Win16 та Win32

Константи, які задають конкретні максимальні і мінімальні значення цілочисельних і дійсних даних в залежності від їх типу, містяться у вигляді макросів в спеціальних заголовних файлах limits.h, float.h, vaiues.h стандартної бібліотеки C++. Ці файли забезпечують переносимість програм на мові C++, тобто програма, розроблена для операційної системи Unix повинна працювати в будь-який інший операційній системі з мінімумом доробок, наприклад, в операційній системі MS DOS або Microsofl Windows (теоретично).

Зазвичай діапазон представлення для цілочисельних даних int залежить від тієї платформи, для якої розробляється програма (Application) – див. табл.4.1 і табл.4.2. Наприклад, для IBM PC за замовчуванням компілятор Borland C++4.x встановлює платформу Winl6 (Windows 3.x), яка називається EasyWin, компілятор Borland C++3.x – MS DOS Standard. Компілятори Borland C ++ 5.x, Visual C ++ 4.x (і вище) – Win32 Console – 32-х розрядний консольний додаток (вікно MS DOS) в операційних системах лінійки Microsoft Windows 9.x / NT / 2000.

Таблиця 4.1

Типи цілочисельних даних платформи Win16

Тип	Довжина (біт)	Діапазон допустимих значень
unsigned char	8	0…255
char	8	-128…127
enum	16	-32768…32767
unsigned int	16	0…65535
short int	16	-32768…32767
int	16	-32768…32767

Продовження табл.4.1

Типи цілочисельних даних платформи Win16

Тип	Довжина (біт)	Діапазон допустимих значень
unsigned long	32	0…4 294 967 295
long	32	-2 147 483 648…2 147 483 647

Таблиця 4.2

Типи цілочисельних даних платформи Win32

Тип	Довжина (біт)	Діапазон допустимих значень
unsigned char	8	0…255
char	8	-128…127
enum	32	-2 147 483 648…2 147 483 647
unsigned int	32	0…4 294 967 295
short int	16	-32768…32767
int	32	-2 147 483 648…2 147 483 647
unsigned long	32	0…4 294 967 295
long	32	-2 147 483 648…2 147 483 647

Регістри процесора

Починаючи з моделі 80386 процесори Intel надають 16 основних регістрів для призначених для користувача програм і ще 11 регістрів для роботи з мультимедійними додатками (MMX) і числами з плаваючою точкою (FPU / NPX). Всі команди так чи інакше змінюють вміст регістрів. Як вже говорилося, звертатися до регістрів швидше і зручніше, ніж до пам'яті. Тому при програмуванні на мові Асемблера регістри використовуються дуже широко.

Розглянемо основні регістри процесорів Intel(табл.4.3). Назви та склад/кількість регістрів для інших процесорів можуть відрізнятися.

Таблиця 4.3

Основні регістри процесора

Назва	Разрядність	Основне призначення
EAX	32	Акумулятор
EBX	32	База
ECX	32	Лічильник
EDX	32	Регістр даних
EBP	32	Покажчик бази
ESP	32	Покажчик стека

Продовження табл. 4.3

Основні регістри процесора

Назва	Разрядність	Основне призначення

ESI	32	Індекс джерела
EDI	32	Індекс приймача
EFLAGS	32	Регістр прапорів
EIP	32	Покажчик інструкції (команди)
CS	16	Сегментний регістр
DS	16	Сегментний регістр
ES	16	Сегментний регістр
FS	16	Сегментний регістр
GS	16	Сегментний регістр
SS	16	Сегментний регістр

Регістри EAX, EBX, ECX, EDX – це регістри загального призначення. Вони мають певне призначення (так уже склалося історично), проте в них можна зберігати будь-яку інформацію.

Регістри EBP, ESP, ESI, EDI – це також регістри загального призначення. Вони мають вже більш конкретне призначення. У них також можна зберігати призначені для користувача дані, але робити це потрібно вже більш обережно, щоб не отримати «несподіваний» результат.

Регістр прапорів і сегментні регістри вимагають окремого опису і будуть більш детально розглянуті далі.

Колись процесори були 16-розрядними, і, відповідно, всі їх регістри були також 16-розрядними. Для сумісності зі старими програмами, а також для зручності програмування деякі регістри розділені на 2 або 4 «маленьких» регістра, у кожного з яких є свої імена. У таблиці 4.4 перераховані такі регістри.

Приклад такого регістра показано на рис.4.1.

З цього випливає, що ви можете написати в своїй програмі, наприклад, такі команди:

MOV AX, 1

MOV EAX, 1

Обидві команди помістять в регістр AX число 1. Різниця буде полягати лише в тому, що друга команда обнулить старші розряди регістра EAX, тобто після виконання другої команди в регістрі EAX буде число 1. А перша команда залишить в старших розрядах регістру EAX старі дані. І якщо там були дані, відмінні від нуля, то після виконання першої команди в регістрі EAX буде якесь число, але не 1. А ось в регістрі AX буде число 1.

<table>
<tr><th>Розряд(біт)</th><th>Регістр(32 біти)</th><th>Регістр(16 біт)</th><th>Регістр(16 біт)</th></tr>
<tr><td>31</td><td rowspan="32">EAX</td><td colspan="2" rowspan="16">Старші розряди регістра EAX</td></tr>
<tr><td>30</td></tr>
<tr><td>29</td></tr>
<tr><td>28</td></tr>
<tr><td>27</td></tr>
<tr><td>26</td></tr>
<tr><td>25</td></tr>
<tr><td>24</td></tr>
<tr><td>23</td></tr>
<tr><td>22</td></tr>
<tr><td>21</td></tr>
<tr><td>20</td></tr>
<tr><td>19</td></tr>
<tr><td>18</td></tr>
<tr><td>17</td></tr>
<tr><td>16</td></tr>
<tr><td>15</td><td rowspan="16">AX</td><td rowspan="8">AH</td></tr>
<tr><td>14</td></tr>
<tr><td>13</td></tr>
<tr><td>12</td></tr>
<tr><td>11</td></tr>
<tr><td>10</td></tr>
<tr><td>9</td></tr>
<tr><td>8</td></tr>
<tr><td>7</td><td rowspan="8">AL</td></tr>
<tr><td>6</td></tr>
<tr><td>5</td></tr>
<tr><td>4</td></tr>
<tr><td>3</td></tr>
<tr><td>2</td></tr>
<tr><td>1</td></tr>
<tr><td>0</td></tr>
</table>

Рисунок 4.1 – Приклад регістра

Нижче наведено список регістрів загального призначення, які можна поділити описаним вище способом і при цьому до «половинкам» і «четвертинки» цих регістрів можна звертатися в програмі як до окремого регістру(табл.4.4).

Таблиця 4.4

Список регістрів загального призначення

Регістр	Старші розряди	Імена 16-ти та 8-ми бітних регістрів

	31…16	15…8	7…0
EAX	…	AX	
		AH	AL
EBX	…	BX	
		BH	BL
ECX	…	CX	
		CH	CL
EDX	…	DX	
		DH	DL
ESI	…	SI	
EDI	…	DI	
EBP	…	BP	
ESP	…	SP	
EIP	…	IP	

Наступні чотири внутрішніх регістри процесора – це сегментні регістри, кожний з яких визначає положення одного з робочих сегментів (рис. 4.2):

1. регістр CS (Code Segment) відповідає сегменту команд, які виконуються в даний момент;
2. регістр DS (Data Segment) відповідає сегменту даних, з якими працює процесор;
3. регістр ES (Extra Segment) відповідає додатковому сегменту даних;
4. регістр SS (Stack Segment) відповідає сегменту стека.

Сегментні регістри використовуються для організації сегментної адресації пам'яті і призначені для зберігання базових адрес поточних сегментів пам'яті.

Регістр стану (англ. Program State Word - PSW, англ. Flag Register - RF, англ. Condition Code Register - CCR) – це частина процесора, що зберігає важливу інформацію про стан обчислювальної системи, наприклад, біти-прапорці, які характеризують результати виконання арифметичних чи логічних операцій та порівнянь. Залежно від архітектурних особливостей, вміст регістра стану може бути частиною так званого контексту, що записується в стек при перериванні, або це покладено на плечі програміста.

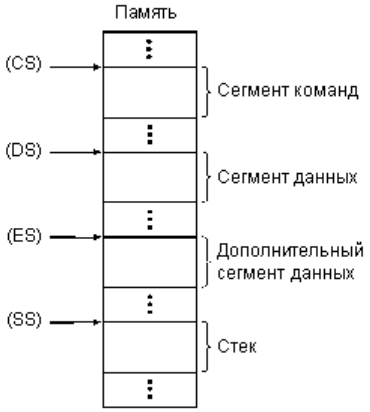

Рисунок 4.2 – Сегменти команд, даних і стека в пам'яті

Часто система команд передбачає спеціальні інструкції для читання та запису бітів регістра стану, оскільки вони застосовуються зі специфічною метою: для зміни природнього порядку слідування команд та управління процесором.

Прапорці необхідні для визначення шляху виконання та використовуються командами переходів(табл.4.5). Наприклад, якщо певна операція дала нульовий результат, виконується одна група інструкцій, інакше – інша. В наведеному нижче прикладі перехід здійснюється за умови активності ознаки нульового результату. В таблиці 4.5 наведено найбільш вживані ознаки процесорів.

Таблиця 4.5

Найчастіше вживані прапорці

Мнемоніка	Назва	Опис
Z	Прапорець нуля (англ. *Zero flag*)	Встановлюється, якщо результат арифметичної чи логічної операції рівний нулю.
C	Прапор переносу (англ. *Carry flag*)	Використовується для додавання/віднімання чисел, більших за розрядну сітку мікропроцесора. Інколи застосовується для збереження витісненого біту при логічних, арифметичних та циклічних зсувах.

Продовження табл. 4.5

Найчастіше вживані прапорці

Мнемоніка	Назва	Опис

S	Прапорець знаку (англ. *Sign flag*)	Дублює старший біт машинного слова, що зазвичай використовується як знак.
N	Прапорець негативного результату (англ.*Negative flag*)	Встановлюється, якщо результат арифметичної операції від'ємний.
O	Прапорець переповнення (англ. *Overflow flag*)	Використовується для позначення ситуації, коли двійкове число занадто велике для збереження в регістрі результату.

Основні команди мови Assembler

Асемблер може працювати з досить широким діапазоном даних. Почнемо ми з найпростіших базових команд цілочисельної арифметики. Дані команди будуть працювати зі знаковими або беззнаковими даними довжиною 8 або 16 біт. Деякі команди зможуть обробляти і 32-розрядні дані (команди пересилання, команди додавання і віднімання).

Основна більшість цих команд НЕ розрізняє знакові і беззнакові дані – це прерогатива людини. Знак "відповідає" найстарший біт числа в його внутрішньому (машинному) поданні. Чи враховувати його як знак або як інформаційну частину числа вирішує людина.

Команди пересилання і обміну інформацією. Це найпростіші і найпоширеніші команди. Насправді, команд пересилань в Асемблері набагато більше, але розглянемо базові команди.

Команди Ассемблера містять мнемокод, покликаний полегшити розуміння людиною даної команди.

1. Команда пересилки MOV

Мнемокод цієї команди отриманий в результаті скорочення такої пропозиції: MOVe operand to/from system registers – Пересилання операнда в/з системних регістрів.

Команда ця містить два операнда і має наступний формат (синтаксис):

MOV Destination, Source

MOV Приймач, Джерело

Це ДУЖЕ поширений формат команд і, якщо команда містить два операнда, вони майже завжди інтерпретуються саме так: перший операнд – приймач, а другий – джерело.

В даному конкретному випадку інформація з джерела пересилається (копіюється) в приймач. Ця команда в найпростішій своїй ін-

терпретації відповідає оператору присвоювання (вміст джерела <Джерело> копіюється в приймач):

Приймач: = <Джерело>

ДОВЖИНА ОБОХ операндів повинна бути однаковима. Якщо ж вона різна, використовується директива вказівки типу:

тип PTR [вираз]

Наприклад, BYTE PTR a + 2

У цій простій на перший погляд команді є винятки:

1. В якості приймача НЕ може бути регістр CS.

2. Не можна записати команду ПРЯМИЙ ініціалізації сегмента даних: DSEG SEGMENT

……..

mov DS, Dseg

У цій ситуації команду MOV можна розписати на дві:

mov AX, Dseg

mov DS, AX

3. Не можна пересилати сегментні регістри:

mov ES, DS

У цій ситуації можна зробити так:

mov AX, DS

mov ES, AX

Аналогічно не можна використовувати і команду Mov DS, ES.

2. Команда обміну даними XCHG

Команда використовується для двунаправленої пересилки даних. Для цієї операції можна, звичайно, застосувати послідовність із декількох команд MOV, але через те, що операція обміну використовується досить часто, розробники системи команд процесора вирішили ввести окрему команду обміну XCHG.

Команда XCHG міняє місцями вміст двох операндів (eXCHanGe).

Синтаксис:

XCHG Приймач, Джерело

Існує три варіанта команди XCHG:

XCHG Register Register

XCHG Register Memory

XCHG Memory Register

Для операндів команди XCHG необхідно дотримуватися тих же правил і обмежень, що і для операндів команди MOV, за виключенням того, що операнди команди XCHG не можуть бути безпосередньо заданими числами.

Приклади використання команди XCHG:

1. Обмін вмісту 16-розрядних регістрів:

xchg AX, BX

2. Обмін вмісту 8-розрядних регістрів:

xchg AH, AL

3. Обмін вмісту 16-розрядного операнда в пам'яті і регістра BX:

xchg VAR1, BX

4. Обмін вмісту 32-розрядних регістрів:

xchg EAX, EBX

Арифметичні команди. Ця група команд дозволяє розібратись, як IBM PC виконує найпростіші арифметичні операції з цілочисельними даними довжиною 8 і 16 біт (частково і з даними довжиною 32 біти). Всі ці команди сигналізують про результат процесу обрахування, заносячи відповідні значення у відповідні біти регістра.

1. Команди додавання

Ці команди використовуються для додавання чисел в двійковій системі числення. Якщо в результаті додавання результат не помістився у відповідне місце, виробляється позначка переносу CF=1. У цьому випадку говорять, що позначка CF встановилася. Є ще 4 позначки(табл.4.6).

Таблиця 4.6

Стан прапорців після виконання команд додавання

Прапорці	Пояснення
CF=1	Результат додавання не помістився в операнд-приймач
PF=1	Результат додавання має парну кількість бітів із значенням 1

Продовження табл. 4.6

Прапорці	Пояснення
SF=1	Копіюється СТАРШИЙ (ЗНАКОВИЙ) біт результату додавання
ZF=1	Результат додавання дорівнює нулю
OF=1	При додаванні двох чисел з ОДНАКОВИМ знаком (два додатні або два від'ємні) результат додавання вийшов БІЛЬШЕ ніж

	допустиме значення. В цьому випадку приймач МІНЯЄ ЗНАК.

1.1 Команда ADD

Найпростіша із команд додавання – ADD (ADDition - додавання).

Синтаксис:

ADD Приймач, Джерело

Логіка роботи команди:

<Приймач> = <Приймач> + <Джерело>

Можливі поєднання операндів для цієї команди аналогічні команді MOV.

Нехай у нас є два числа – 120 і 100. Щоб їх додати. Необхідно виконати наступні дії:

```
mov   al, 120     ;    al <=== 120
mov   cl, 100     ;    cl <=== 100
add   al, cl      ;    <al>:=<al>+<cl>
mov   x, al       ;    x  <=== <al>
```

1.2 Команда INC

Мнемокод цієї команди отриманий в результаті скорочення такого речення: INCrement operand by 1 – Збільшення значення операнда на 1. Ця команда містить один операнд і має наступний синтаксис:

INC Операнд

Логіка роботи команди:

<Операнд> = <Операнд> + 1

В якості операнда допустимі регістри і пам'ять: R8, R16, Mem8, Mem16. Наприклад:

```
inc I    ;  I++
inc ax   ;  <ax> = <ax> + 1
inc cl   ;  <cl> = <cl> + 1
```

2. Команди віднімання

Ці команди обернені відповідним командам додавання. Вони мають ті ж операнди.

2.1 Команда SUB

Мнемокод цієї команди отриманий в результаті скорочення слова SUBstraction – віднімання. Її синтаксис:

SUB Приймач, Джерело

При виконанні віднімання треба бути особливо уважним до стану позначок, тому що коли від одного числа віднімається інше, можливість отримати від'ємне число в якості результата суттєво більша. Тому потрібно відслідковувати стан позначки SF і, якщо вона буде встановлена, то приймати необхідні міри, якщо в цьому буде необхідність.

Приклад використання команди:

```
mov  al, 10
sub  al, 7    ;     al = 3; al – приймач, 7 – джерело.
```

1.1 Команда DEC

Команда DEC зменшує на одиницю вміст приймача. Вона еквівалентна команді:

SUB Приймач, 1

Логіка роботи команди:

<Операнд> = <Операнд> - 1

Приклад використання команди:

```
mov    al, 15
dec     al;    al = 14
```

2.2 Команда NEG

Команда NEG (NETate operating) – зміна знака операнда. Її синтаксис:

NEG Операнд

Логіка роботи команди:

<Операнд> = - <Операнд>

В якості операнда допустимі регістри R8, R16, Mem8, Mem16. Наприклад, для обчислення <AL> = =50 - <AL> більш доцільно використати такий код:

```
neg   al
add   al, 50
```

Як бачимо, операцію віднімання замінили додаванням. Це пов'язано з тим, що операція віднімання повільніша, ніж операція додавання.

Приклад: Написати програму для обчислення заданого цілочисленого виразу –(b+c+d)–(a+1) для початкових даних в знаковому форматі довжиною 8 біт: ShortInt (signed char), використовуючи арифметичні операції ADD, INC, SUB, DEC, NEG.

Вхідними даними будуть змінні a, b, c, d. Результуючими змінними будуть res_c(результат обчислення на мові C++) та res_asm(результат обчислення на мові Assembler).

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include <cstdlib>

int main(int argc, _TCHAR* argv[])
{
      signed char a, b, c, d, res_c, res_asm;
      printf("a = "); scanf_s("%d", &a);
      printf("b = "); scanf_s("%d", &b);
      printf("c = "); scanf_s("%d", &c);
      printf("d = "); scanf_s("%d", &d);
      res_c = -(b + c + d) - (a + 1);
      printf("Result C = %d\n", res_c);
      __asm {
              mov al, b;        // <al> = b
              add al, c;        // <al> = b + c
              add al, d;        // <al> = b + c + d
              neg al;           // <al> = -(b + c + d)
              mov cl, a;        // <cl> = a
              inc cl;           // <cl> = a + 1
              sub al, cl;       // <al> = -(b + c + d) - (a + 1)
              mov res_asm, al;  // res_asm = al
      }
      printf("Result ASM= %d\n", res_asm);
      system("Pause");
return 0;
    }
```

Проведемо тестові перевірки роботи програми для даних, що не перевищують та що перевищують діапазон допустимих значень відповідного формату(рис.4.3 – рис.4.4). Крім того, запишемо значення регістрів при покроковому виконанні.

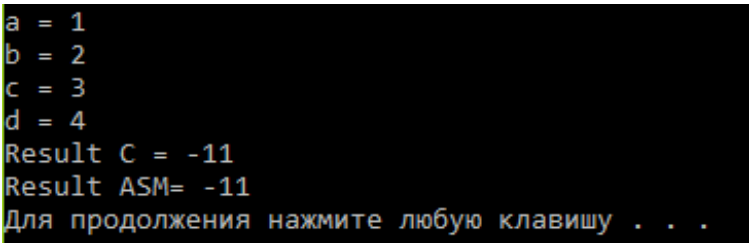

Рисунок 4.3 – Перевірка роботи програми для 8-бітних вхідних даних та 8-бітного результату

Значення регістрів при покроковому виконанні

Крок	Команда	Значення регістра				EFLAGS/FLAGS (CF, OF)
		al	ah	cl	ch	
1	mov al, b	2	н/в	н/в	н/в	–
2	add al, c	5	н/в	н/в	н/в	–
3	add al, d	9	н/в	н/в	н/в	–
4	neg al	-9	н/в	н/в	н/в	–
5	mov cl, a	н/в	н/в	1	н/в	–
6	inc cl	н/в	н/в	2	н/в	–
7	sub al, cl	-11	н/в	н/в	н/в	–
8	mov res_asm, al	-11	н/в	н/в	н/в	–

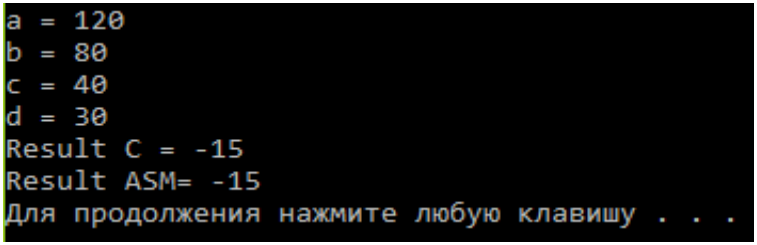

Рисунок 4.4 – Перевірка роботи програми для 8-бітних значень вхідних даних та результату, що перевищує 8 біт

Значення регістрів при покроковому виконанні

Крок	Команда	Значення регістра				EFLAGS/FLAGS (CF, OF)
		al	ah	cl	ch	
1	mov al, b	80	н/в	н/в	н/в	–
2	add al, c	120	н/в	н/в	н/в	–
3	add al, d	150	н/в	н/в	н/в	–

4	neg al	-150	н/в	н/в	н/в	–
5	mov cl, a	н/в	н/в	120	н/в	–
6	inc cl	н/в	н/в	121	н/в	–
7	sub al, cl	-15	н/в	н/в	н/в	1
8	mov res_asm, al	-15	н/в	н/в	н/в	1

В останньому випадку результат не є коректним, адже відбувається переповнення CF.

Завдання на лабораторну роботу

1. Написати програму для обчислення заданого цілочисленого виразу(табл.4.7) для початкових даних в знаковому форматі довжиною 8 біт, використовуючи арифметичні операції ADD, INC, SUB,

DEC, NEG. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Таблиця 4.7

Дані для розрахунку

№ Варіанту	Вираз
1	–(a+b–c)–(d+1)–(e–f)
2	1–(b–d+c)+(a+f)–e+2
3	–(b+c+d–f)–(a+1–e)–1
4	f–(a+c–b+e)–(d+2)+3
5	–(b–1+c–1)+(a–e+d+f)
6	(b–1)+(a+1)–c–d–(f–e)
7	12–(b–c–a)+d–(e+1)+f
8	(a+b–e)–1–(c+d)–(f+3)
9	–(a+b+c–1–d)–(f+e+1)
10	–(a+b–c–d)+2+e–f+2
11	b+c+d–(a+4)–(e+1)+f
12	c+d–(a+b+1)–(f+1–e)
13	(d–c)+(b–a)–1+(e–1+f)
14	(d+c–1)+(a+b–1)–(f–e)
15	–(a+b–1)+(c–d)+e–f+6
16	(a+b)+3–(d–c–1)–(e+f)
17	(b–1)+(a+1)–c+d–(f+e)
18	–(a+c–b)–(d–1)+4–f+e
19	(a+1)+b+c–d–2–(f+1–e)
20	b+c+d–a+2–(e+1)–(f–1)
21	–(a+1)–b+c+d–f–(e+1)
22	–(b+2)–c+a–d+e–(f+1)
23	–(c–2)+a+b+d–(e+1)–f
24	–(d+2)+a–e–b+c+(f–1)
25	–(a+b)–(d–c+1)+(f–e)–1
26	a+1–(c+d–b)+(f+1–e)
27	c–1–(a–b+d)–(e–1)+f
28	c–3+a+b–(d+1)+e–(f+1)
29	–(2+a–b)+c+d–(e+1)+f
30	–(b–a)+c+d+2–(f–1+e)

2. Написати програму для обчислення заданого цілочисленого виразу(табл.4.7) для початкових даних в знаковому форматі довжиною 16 біт, використовуючи арифметичні операції ADD, INC, SUB, DEC, NEG. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

3. Написати програму для обчислення заданого цілочисленого виразу(табл.4.7) для початкових даних в знаковому форматі довжиною 32 біт, використовуючи арифметичні операції ADD, INC, SUB, DEC, NEG. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Контрольні питання

1. Типи цілочисельних даних платформ Win16 та Win32.
2. Регістри загального призначення. Призначення та функції.
3. Сегментні регістри. Призначення та функції.
4. Регістри стану та керування.
5. Основні команди мови Assembler.
6. Команда пересилки MOV. Призначення та синтаксис.
7. Команда обміну даними XCHG. Призначення та синтаксис.
8. Команди додавання. Призначення та синтаксис.
9. Команди віднімання. Призначення та синтаксис.
10. Стан позначок після виконання команд додавання.

Лабораторна робота № 5
ОБЧИСЛЕННЯ СКЛАДНИХ ЦІЛОЧИСЛЕНИХ ВИРАЗІВ НА МОВІ ASSEMBLER

Мета заняття: ознайомитися з основними командами мови Assembler для обчислення складних цілочисельних виразів; набути практичних навичок в написанні програм для обчислення складних цілочисельних виразів на мові Assembler.

Теоретичні відомості

Основні команди мови Assembler

Асемблер може працювати з досить широким діапазоном даних. Розглняемо складні команди цілочисельної арифметики. Дані команди будуть працювати зі знаковими або беззнаковими даними довжиною 8 або 16 біт. Деякі команди зможуть обробляти і 32-розрядні дані.

Основна більшість цих команд НЕ розрізняє знакові і беззнакові дані - це прерогатива людини. Знак "відповідає" найстарший біт числа в його внутрішньому (машинному) поданні. Чи враховувати його як знак або як інформаційну частину числа вирішує людина.

Арифметичні команди. Ця група команд дозволяє розібратись, як IBM PC виконує найпростіші арифметичні операції з цілочисельними даними довжиною 8 і 16 біт (частково і з даними довжиною 32 біти). Всі ці команди сигналізують про результат процесу обрахування, заносячи відповідні значення у відповідні біти регістра.

1. Команди множення

Команди множення MUL (MULtiply – беззнакове множення) та IMUL (Integer MULtiply – знакове цілочисельне множення) містять неявні операнди. Як видно із визначення команд, вони враховують наявність знака.

Синтаксис:

MUL Джерело

IMUL Джерело

Логіка роботи команд:

<Добуток>=<Множник>*<Джерело_Множник>

В якості Джерела_Множника допустимі регістри і пам’ять: R8, R16, Mem8, Mem16. Константи не допускаються!

Добуток і множник знаходяться в строго визначеному місці в залежності від довжини Джерела.

Таблиця 5.1

Неявні операнди команд MUL, IMUL

Довжина джерела	Множник	Добуток
Byte	AL	AX (<AH:AL>)
Word	AX	<DX:AX>

Довжина Добутку завжди в два рази більша, ніж у Множника. Причому старша частина Добутку знаходиться або в регістрі AH, або в DX.

Ці команди також виробляють прапорці CF=1 та OF=1. Але вони встановлюються тільки тоді, коли результат занадто великий для відведених йому регістрів призначення.

Множення для 8-розрядних (знакових і беззнакових даних), а також для 16-розрядних знакових по ідеї абсолютно аналогічне.

3. Команди ділення

DIV (DIVide – беззнакове ділення), IDIV(Integer DIVide – знакове ділення цілих цисел).

Синтаксис:

DIV Джерело

IDIV Джерело

Логіка роботи команди:

<Частка:Залишок>=<Ділене>/<Джерело_Дільник>

Отже, Джерелом є Дільник. В якості Дільника допустимі регістри і пам’ять: R8, R16, Mem8, Mem16. Константи не допускаються!

Ділене і <Частка:Залишок> знаходяться в строго визначеному місці в залежності від довжини Дільника.

Таблиця 5.2

Неявні операнди команд DIV, IDIV

Довжина джерела (Дільника)	Ділене	Добуток	
		Частка	Залишок
Byte	AX(<AH:AL>)	AL	AH
Word	<DX:AX>	AX	DX

Довжина діленого завжди в два рази більша, ніж у Дільника. Причому старша частина Діленого знаходиться в регістрі AH, або в DX.

Ці команди також виробляють прапорці CF=1 та OF=1. Але вони встановлюються коли частка не поміщається в регістри AL або AX. Тут, на відміну від команд множення, завжди генерується переривання «Ділення на нуль», хоча на нуль і не ділимо.

3. Команди розширення

Беззнакові числа займають всю комірку пам'яті, поняття знак для них не існує – вони вважаються додатніми. Тому при діленні беззнакових чисел в старшу частину діленого треба занести нуль. Це можна зробити уже відомими нам командами: mov ah,0 або mov dx,0.

Але це не ефективно, тому ми використовуємо рідну для комп'ютера команду додавання по модулю 2 – xor ah,ah або xor dx, dx.

Для знакових даних існують дві команди розширення знака. CBW (Convert Byte to Word – перетворити байт, який знаходиться в регістрі AL, в слово – регістр AX) і CWD (Convert Word to Double word – перетворити слово, яке знаходиться в регістрі AX в подвійне слово – регістри <DX:AX>). Операнди їм не потрібні.

Синтаксис:

CBW

CWD

Таблиця 5.3

Логіка роботи команди CBW при отриманні знакового діленого

1) Отримуєм старшу частину (AH)	**Відома молодша частина (AL)**	
Біти 15-8	**Знак – біт 7**	**Біти 6-0**
1111 1111	1	Інформаційна частина числа
0000 0000	0	Інформаційна частина числа
2) Отримуємо Знакове ділене – регістр AX (AH:AL)		

Ці команди також застосовуються при вирівнюванні операндів по довжині.

Таблиця 5.4

Логіка роботи команди CWD при отриманні знакового ділено

1) Отримуєм старшу частину (DX)	**Відома молодша частина (AX)**	
Біти 31-16	**Знак – біт 15**	**Біти 14-0**
1111 1111 1111 1111	1	Інформаційна частина числа
0000 0000 0000 0000	0	Інформаційна частина числа
2) Отримуєм Знакове ділене – регістр <DX:AX>		

Команда CWDE (Convert Word to Double Extended) виконує перетворення значення, яке знаходиться в регістрі AX до розміру подвійного слова і поміщає його в регістр EAX. При цьому вільні старші біти EAX заповнюються знаковим бітом AX (AX ->EAX).

Команда CDQ (Convert Double to Quadruple) виконує перетворення значення, яке знаходиться в регістрі EAX до розміру зчетвереного слова і поміщає його в пару регістрів EDX:EAX. При цьому вільні старші біти регістра EDX заповнюються знаковим бітом EAX (EAX -> <EDX:EAX>).

Приклад: Написати програму для обчислення заданого цілочисленого виразу (c/4–d*62)/(a*a+1) для початкових даних в знаковому форматі довжиною 8 біт: ShortInt (signed char), використовуючи арифметичні операції ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Вхідними даними будуть змінні a, c, d. Результуючими змінними будуть res_c(результат обчислення на мові C++) та res_asm(результат обчислення на мові Assembler).

Обчислення почнемо з чисельника. Потім розрахуємо знаменник. В чисельнику є дія ділення **(c/4)**. Тоді регістр, який приймає значення **c**, повинен бути 16-бітним. Результат ділення буде 8-бітним.

При множенні використовуємо регістр **al**, який необхідно розширити до регістра **ax**. Чисельник повинен бути 16-бітним, а знаменник 8-бітним.

Результат обчислення заданого виразу(зміна res_asm) буде 8-бітним. Допустимий діапазон значень для вхідних даних та результату: ShortInt (signed char) -128…127.

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include <cstdlib>

int _tmain(int argc, _TCHAR* argv[])
{
	signed char a, c, d, res_c, res_asm;
	printf("(c/4-d*62)/(a*a+1)\n");
	printf("Enter the values of the range [-128...127]\n");
	printf("a = "); scanf_s("%d", &a);
	printf("c = "); scanf_s("%d", &c);
	printf("d = "); scanf_s("%d", &d);
	res_c = (c / 4 - d * 62) / (a*a + 1);
	printf("Result C = %d\n", res_c);

	__asm {
		mov al, c;		// <al> = c
		cbw;			// <ax> = <al>
		mov bl, 4;		// <bl> = 4
		idiv bl;		// <al> = c/4
		mov bl, al;		// <bl> = c/4
		mov al, d;		// <al> = d
		mov cl, 62;		// <cl> = 62
		imul cl;		// <ax> = d*62
		mov cl, 1;		// <cl> = 1
		idiv cl;		// <al> = d*62
		mov dl, al;		// <dl> = d*62
		sub bl, dl;		// <bl> = c/4-d*62
		mov al, a;		// <al> = a
		imul al;		// <ax> = a*a
		inc ax;			// <ax> = a*a+1
		idiv cl;		// <al> = a*a+1
		mov cl, al;		// <cl> = a*a+1
		mov al, bl		// <al> = c/4-d*62
		cbw;			// <ax> = c/4-d*62
		idiv cl;		// <al> = (c/4-d*62)/(a*a+1)
```

```
        mov res_asm, al; // res =  <al>
    }
    printf("Result ASM = %d\n", res_asm);
    system("Pause");
    return 0;
}
```

Проведемо тестову перевірку роботи програми(рис. 5.1)

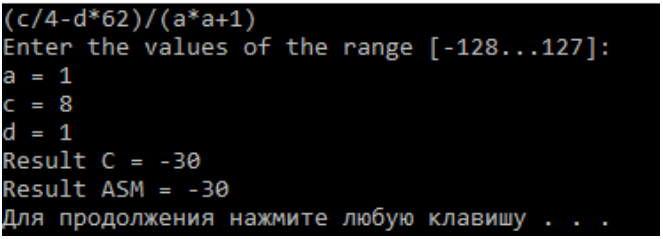

Рисунок 5.1 – Перевірка роботи програми для даних 8-бітних даних та 8-бітного результату

Значення регістрів при покроковому виконанні

Крок	Команда	Значення регістра					EF-LAGS/FLAGS (CF, OF)
		al	ax	bl	cl	dl	
1	mov al, c	8	н/в	н/в	н/в	н/в	–
2	cbw	н/в	8	н/в	н/в	н/в	
3	mov bl, 4	н/в	н/в	4	н/в	н/в	–
4	idiv bl	2	н/в	н/в	н/в	н/в	–
5	mov bl, al	н/в	н/в	2	н/в	н/в	–
6	mov al, d	1	н/в	н/в	н/в	н/в	–
7	mov cl, 62	н/в	н/в	н/в	62	н/в	–
8	imul cl	н/в	62	н/в	н/в	н/в	–
9	mov cl, 1	н/в	н/в	н/в	1	н/в	–
10	idiv cl	62	н/в	н/в	н/в	н/в	–
11	mov dl, al	н/в	н/в	н/в	н/в	62	–
12	sub bl, dl	н/в	н/в	-60	н/в	н/в	–
13	mov al, a	1	н/в	н/в	н/в	н/в	–
14	imul al	н/в	1	н/в	н/в	н/в	–
15	inc ax	н/в	2	н/в	н/в	н/в	–
16	idiv cl	2	н/в	н/в	н/в	н/в	–
17	mov cl, al	н/в	н/в	н/в	2	н/в	–
18	mov al, bl	-60	н/в	н/в	н/в	н/в	–
19	cbw	н/в	-60	н/в	н/в	н/в	–
20	idiv cl	-30	н/в	н/в	н/в	н/в	–

Введемо вхідні данні, які виходять за межі допустимого діапазону значень(рис. 5.2).

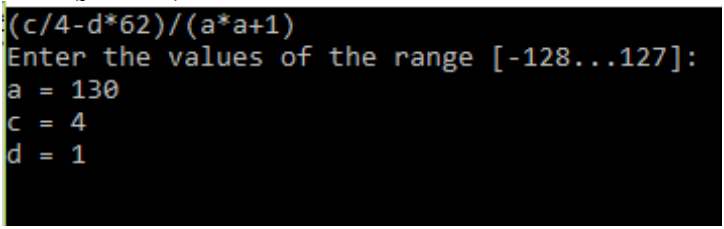

Рисунок 5.2 – Перевірка роботи програми для даних, що виходять за межі діапазону допустимих значень

В результаті виникає помилка(рис. 5.3).

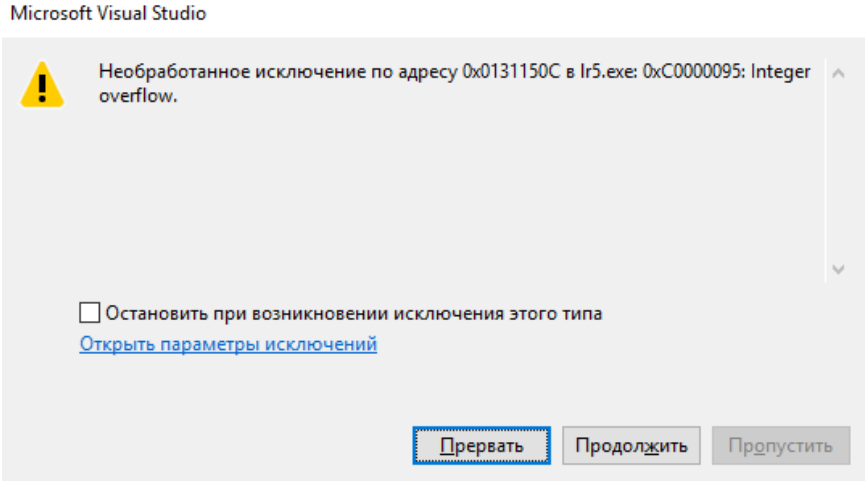

Рисунок 5.3 – Помилка при введенні даних, що виходять за межі діапазону допустимих значень

Отже, необхідно задати умову для перевірки введених даних.

Також, можливий випадок, коли результат розрахунку виходить за межі допустимого діапазону значень.

Завдання на лабораторну роботу

1. Написати програму для обчислення заданого цілочисленого виразу(табл. 5.5) для початкових даних в знаковому форматі довжиною 8 біт, використовуючи арифметичні операції ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

2. Написати програму для обчислення заданого цілочисленого виразу(табл. 5.5) для початкових даних в знаковому форматі довжиною 16 біт, використовуючи арифметичні операції ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

3. Написати програму для обчислення заданого цілочисленого виразу(табл. 5.5) для початкових даних в знаковому форматі довжиною 32 біт, використовуючи арифметичні операції ADD, ADC, INC, SUB, SBB, DEC, NEG, IMUL, IDIV, CBW, CWD. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Таблиця 5.5

Дані для розрахунку

№ Варіанту	Вираз
1	(c+4*d-123+e)/(1-a/2+f)
2	(2*c+d-52+e)/(a/4+1-f)
3	(-2*c-d+53-e)/(a/4-1+f)
4	(2+c-d*23-e)/(2*a*a-1-f)
5	(4*c+d-1+e)/(c-a/2+f)
6	(25/c-d+2+e)/(b+a*a-1-f)
7	(4*c-d/2+23-e)/(b+a*a-1+f)
8	(c/d+3*a/2-e)/(c-a+1-f)
9	(2*c+4/d+23+e)/(a*a-1+f)
10	(2*c/d+2+e)/(d-a*a-1-f)
11	(2*c/a-d*d-e)/(d+a-1+f)
12	(-15*a+b-a/4-e)/(b*a-1-f)

Продовження табл. 5.5

Дані для розрахунку

№ Варіанту	Вираз
13	(4*a-1+b/2+e)/(b*c-5+f)
14	(4*a-b-1+e)/(c/b+a-f)
15	(b+c*b-a/4-e)/(a*b-1+f)
16	(c/b-24+a-e)/(2*a*c-1-f)
17	(41-d/4-1+e)/(c/b+a*d+f)
18	(b/a+4*c+e)/(c-b+1-f)
19	(c-33+b/4-e)/(a*c/b-1+f)
20	(c/4+28*d-e)/(a/d-c-1-e)
21	(1+6*a-b/2+e)/(c+a/d+f)
22	(4*b-c*a+e)/(b+c/28-1-f)
23	(4/c+3*a-e)/(c/a-b-1+f)
24	(4*a/b+1-e)/(c*b-18+a-f)
25	(b-c/b+a/4+e)/(a*b-1+f)
26	(c*b-24+a+e)/(b/2*c-1-f)
27	(41*d/4+1-e)/(a-c/b+a*d+f)
28	(b*a+c/2-e)/(4*c-b+1-f)
29	(c+23-b*4+e)/(a*c/b-1+f)
30	(c*4+28/d-e)/(a*d-c-1+f)

Контрольні питання

1. Назвіть складні команди цілочисельної арифметики мови Assembler.
2. Команда множення MUL. Призначення та синтаксис.
3. Команда множення IMUL. Призначення та синтаксис.
4. Команда ділення DIV. Призначення та синтаксис.
5. Команда ділення IDIV. Призначення та синтаксис.
6. Команда розширення знаку CBW. Призначення та синтаксис.
7. Команда розширення знаку CWD. Призначення та синтаксис.
8. Неявні операнди команд MUL, IMUL.
9. Неявні операнди команд DIV, IDIV.
10. Логіка роботи команди CBW при отриманні знакового діленого.

Лабораторна робота № 6
ОРГАНІЗАЦІЯ УМОВНИХ ПЕРЕХОДІВ

Мета заняття: ознайомитися з основними командами мови Assembler дляорганізації умовних переходів; набути практичних навичок в написанні програм з організацією умовних переходів на мові Assembler.

Теоретичні відомості

Основні команди мови Assembler для організації умовних переходів

Команди передачі управління дозволяють змінити природню послідовність виконання команд шляхом зміни вмісту регістра IP.

Базові команди передачі управління – це команди безумовного переходу на відповідну адресу в пам'яті комп'ютера, команди умовного переходу в залежності від результату перевірки умови і команди організації циклічних обчислень. Команди передачі управління не змінюють значення прапорців.

1. Команди безумовної передачі управління

Найбільш поширена команда безумовного переходу – команда JMP. Вона є аналогом відомого в алгоритмічних мовах оператора: goto мітка**.**

Мнемокод команди JMP отриманий в результаті скорочення слова JuMP – стрибок. Ця команда містить один операнд і має наступний формат:

JMP мітка

Мітка, як і в алгоритмічних мовах, відмічає потрібну команду, на яку потрібно передати управління, і може мати атрибут NEAR (по замовчуванню) або FAR. Якщо мітка має атрибут NEAR, безумовний перехід здійснюється в межах даного сегмента. В цьому випадку логіка роботи команд наступна:

<IP> = <IP> + зміщення_до_потрібної_команди

Таким чином, автоматично відбувається виконання команди, відміченої міткою.

Оскільки в даному випадку перехід здійснюється в межах сегменту зміщення_до_потрібної_команди не може перевищувати двох байтів. Необхідна нам команда може розміщуватися в сторону більших адрес, тоді говорять, що перехід здійснюється вперед. В

цьому випадку зміщення_до_потрібної_команди – додатнє. Якщо навпаки, необхідна нам команда розміщується в сторону менших адрес, говорять про перехід назад. В цьому випадку зміщення_до _потрібної_команди – від'ємне.

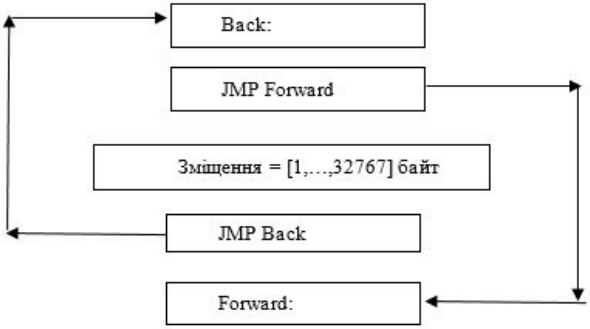

Рис.6.1 – Схема роботи команд JMP Forward (перехід вперед) і JMP Back (перехід назад) в межах одного сегменту

Можна зекономити один байт при реалізації команди безумовної передачі управління, якщо зробити перехід коротким (SHORT), тобто зміщення в межах [-128…127]. В цьому випадку перехід вперед потрібно записати таким чином: JMP SHORT Forward.

Для передачі управління назад слово SHORT можна не писати, тому що компілятор при побудові машинного коду сам визначить, яке вийде зміщення, оскільки на момент компіляції команда переходу JMP Back адреса, на яку вказує мітка Back, йому уже відома (перегляд початкового модуля відбувається зверху вниз). По замовчуванню компілятор робить одне проходження.

2. Команди умовної передачі управління Jcc

Базових команд умовного переходу всього 17, але вони можуть мати різну мнемоніку, тому в загальному виходить 31 команда. Читати їх досить просто, якщо знаєш ключ.

Перша буква команди J від уже відомого нам слова (Jump - стрибок). Інші букви (cc) в скороченому вигляді описують умову переходу. По цій ознаці всі команди умовного переходу можна розбити на три групи.

Перша група команд умовного переходу:

1. E – Equal (дорівнює).

2. N – Not (не, заперечення).

3. G – Greater (більше) – застосовується для чисел із знаком.

4. L – Less (менше) - застосовується для чисел із знаком.

5. A – Above (вище, більше) – застосовується для чисел без знака.

6. B – Below (нрижче, менше) - застосовується для чисел без знака.

Наприклад, команда JL означає перехід, якщо менше. Їй еквівалентна команда-синонім JNGE – перехід, якщо не більше і не дорівнює. Різниця в командах переходу для знакових і беззнакових даних пояснюється тим, що вони реагують на різні прапорці (для знакових даних суттєвий прапорець SF, а для беззнакових - CF).

Умовний перехід для цієї групи команд зазвичай реалізується в два кроки:

1. Порівняння, в результаті чого формуються прапорці (регістр FLAGS).

2. Умовна передача управління (Jcc Коротка_мітка) на відмічену команду в залежності від значення прапорців.

Ця група операцій найчастіше виконується в парі з командою порівняння CMP.

CMP приймач, джерело

Jcc Коротка_мітка

Команда CMP (CoMPare operands) – порівняння операндів. В результаті її виконання утворюються прапорці, які потім аналізуються командами умовного переходу. За станом прапорців зручно спостерігати під час покрокового виконання.

Таблиця 6.1

Команди умовного переходу групи 1

Умова порівняння (CMP)	**Мнемокод**	**Стан прапорців**
Для будь-яких чисел і кодів		
Приймач=джерело	**JE**	**ZF=1**
Приймач<>джерело	**JNE**	**ZF=0**
Для чисел із знаком		
Приймач<джерело	**JL/JNGE**	**SF<>OF**
Приймач<=джерело	**JLE/JNG**	**SF<>OF or ZF=1**

Продовження табл. 6.1

Команди умовного переходу групи 1

Умова порівняння	**Мнемокод**	**Стан прапорців**

(CMP)		
Для чисел із знаком		
Приймач>джерело	**JG/JNLE**	**SF=OF and ZF=0**
Приймач>=джерело	**JGE/JNL**	**SF=OF**
Для чисел без знаку		
Приймач<джерело	**JB/JNAE**	**CF=1**
Приймач<=джерело	**JBE/JNA**	**CF=1 or ZF=1**
Приймач>джерело	**JA/JNBE**	**CF=0 and ZF=0**
Приймач>=джерело	**JAE/JNB**	**CF=0**

Друга група команд умовного переходу реагує на те чи інше значення конкретного прапорця. Тому в мнемоніці даної групи команд завжди вказується перша літера перевіряємого прапорця. Ці команди не потребують обов'язкової наявності команд порівняння перед своєм виконанням. Їм досить будь-якої команди, яка виробляє потрібний прапорець.

Таблиця 6.2

Команди умовного переходу групи 2

Мнемокод	Стан прапорців	Мнемокод	Стан прапорців
JZ/JE	**ZF=1**	**JNZ/JNE**	**ZF=0**
JS	**SF=1**	**JNS**	**SF=0**
JC/JB/JNAE	**CF=1**	**JNC/JAE/JNB**	**CF=0**
JO	**OF=1**	**JNO**	**OF=0**
JP	**PF=1**	**JNP**	**PF=0**

В третю групу команд умовного переходу входить всього одна команда JCXZ. Вона, на відміну від попередніх команд, перевіряє не прапорці, а стан регістра CX на нуль (Jump if CX is Zero).

Синтаксис:

JCXZ Коротка_мітка

Логіка роботи команди:

If (CX=0) then goto Коротка_мітка

Це дуже важлива команда, яка використовується при організації циклів.

Приклад: Написати програму для обчислення заданого умовного цілочисельного виразу для 16-бітних даних, використовуючи команди порівняння, умовного і безумовного переходів. Результат X –

теж цілочисельний і його діапазон (формат) залежить від специфіки вирішуваного умовного виразу. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Дано:

$$X = \begin{cases} 52 * b / a + b, & якщо \quad a > b, \\ -125, & якщо \quad a = b, \\ (a-5)/b, & якщо \quad a < b; \end{cases}$$

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include "windows.h"
#include <stdlib.h>
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[])
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	short a, b, x_c, x_asm;
	printf("Введіть значення з діапазону [-32768...32767]:\n");
	printf("a = "); scanf_s("%hd", &a);
	printf("b = "); scanf_s("%hd", &b);

	if (a > b)
	{
		x_c = (52 * b) / a + b;
	}

	else if (a < b)
	{
		x_c = (a - 5) / b;
	}

	else
	{
		x_c = -125;
	}

	printf("Результат на мові C++ X = %hd\n", x_c);

	__asm
	{
```

```
            mov ax, a       // <ax> = a
            mov bx, b       // <bx> = b
            cmp ax, bx      // порівнюємо значення регістрів <ax> та
                            <bx>

            jg mark1        // a > b
            je mark2        // a = b
            jl mark3        // a < b

            mark1:
                    xchg ax, bx     // міняємо місцями значення
                                    регістрів <ax> та <bx>
                    mov cx, 52      // <cx> = 52
                    imul cx         // <eax> = 52*b
                    idiv bx         // <ax> = (52*b)/a
                    add ax, b       // <ax> = (52*b)/a+b
                    mov x_asm, ax   // x = (52*b)/a+b
                    jmp exit1       // переходимо до мітки exit1

            mark2:
                    mov x_asm, -125 // x_asm = -125
                    jmp exit1       // переходимо до мітки exit1

            mark3:
                    sub ax, 5       // <ax> = a-5
                    cwd             // <dx:ax> = a-5
                    idiv bx         // <ax> = (a-5)/b
                    mov x_asm, ax   // x_asm = (a-5)/b
                    jmp exit1       // переходимо до мітки exit1

            exit1:
    }

    printf("Результат на мові Assembler X = %hd\n", x_asm);
    system ("Pause");
    return 0;
}
```

Проведемо тестову перевірку роботи програми(рис.6.2 – рис.6.4)

```
Введіть значення з діапазону [-32768...32767]:
a = 10
b = 2
Результат на мові C++ X = 12
Результат на мові Assembler X = 12
Для продолжения нажмите любую клавишу . . .
```

Рисунок 6.2 – Перевірка роботи програми для значення a > b

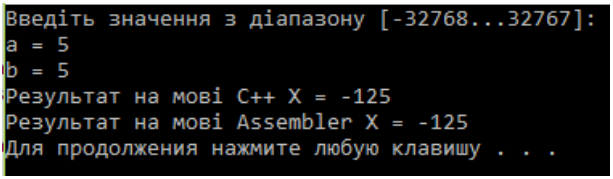

Рисунок 6.3 – Перевірка роботи програми для значення a = b

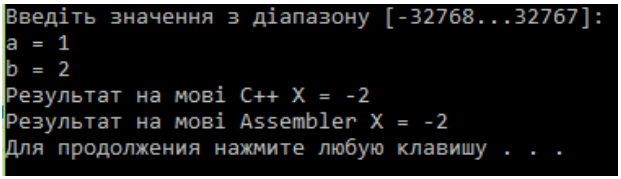

Рисунок 6.4 – Перевірка роботи програми для значення a < b

Значення регістрів при покроковому виконанні

Команда	**Значення регістра**				**EFLAGS/FLAGS (CF, OF)**
	ax	**bx**	**cx**	**<ax:dx>**	
mov ax, a	a	н/в	н/в	н/в	–
mov bx, b	н/в	b	н/в	н/в	–
cmp ax, bx	Порівнюються регістри ax і bx				
jg mark1	Якщо a>b, то переходимо на мітку mark1				
jg mark2	Якщо a=b, то переходимо на мітку mark2				
jg mark3	Якщо a<b, то переходимо на мітку mark3				
mark1:	Мітка mark1(код буде виконуватись в цій мітці, якщо умова – істина)				
xchg ax, bx	н/в	a	н/в	н/в	–
mov cx, 52	н/в	н/в	52	н/в	–
imul cx	н/в	н/в	н/в	52*b	1
idiv bx	(52*b)/a	н/в	н/в	н/в	–
add ax, b	(52*b)/a+b	н/в	н/в	н/в	–
mov x_asm, ax	н/в	н/в	н/в	н/в	–
jmp exit:	Перейти на мітку exit(вихід з ассемблерної вставки)				
mark2:	Мітка mark2(код буде виконуватись в цій мітці, якщо умова – істина)				
mov x_asm, -125	н/в	н/в	н/в	н/в	–
jmp exit:	Перейти на мітку exit(вихід з ассемблерної вставки)				
mark3:	Мітка mark3(код буде виконуватись в цій мітці, якщо умова – істина)				

sub ax, 5	a-5	н/в	н/в	н/в	–
Cwd	н/в	н/в	н/в	a-5	1
idiv bx	(a-5)/b	н/в	н/в	н/в	–
mov x_asm, ax	н/в	н/в	н/в	н/в	–
jmp exit:	Перейти на мітку exit(вихід з ассемблерної вставки)				
exit:	exit(вихід з ассемблерної вставки)				

Введемо вхідні данні, які виходять за межі допустимого діапазону значень(рис. 6.5).

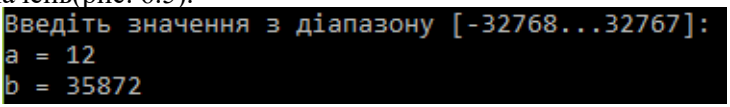

Рисунок 6.5 – Введення даних, які виходять за межі допустимого діапазону значень

В результаті виникає помилка(рис. 6.6).

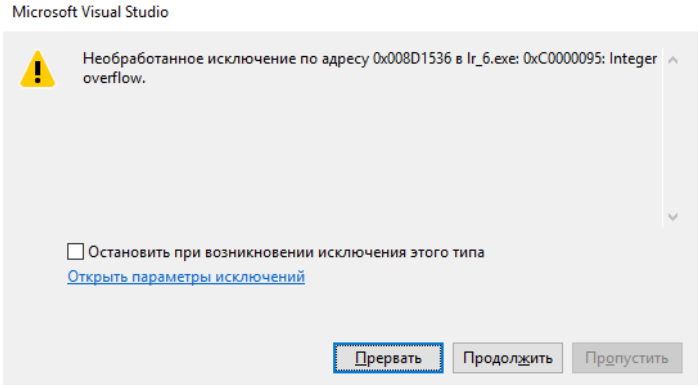

Рисунок 6.6 – Перевірка роботи програми для значень вхідних даних, які виходять за межі допустимого діапазону значень

Отже, необхідно задати умову для перевірки введених даних.

Крім того, необхідно здійснити перевірку ділення на нуль та переповнення.

В даному умовному виразі переповнення може виникнути у двох випадках: в результаті обчислення виразу 52*b/a+b, якщо a>b та в результаті обчислення чисельника виразу (a-5)/b, якщо a<b.

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include "windows.h"
#include <stdlib.h>
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[])
{
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        short a, b, x_c, x_asm, er=0, over=0;
        printf("Введіть значення з діапазону [-32768...32767]:\n");
        printf("a = "); scanf_s("%hd", &a);
        printf("b = "); scanf_s("%hd", &b);

        if (a > b)
        {
                if (a == 0)
                {
                    printf("Помилка: ");
                }

                else
                {
                   x_c = (52 * b) / a + b;
                   if (x_c < -32768 || x_c > 32767)
                   {
                       printf("Помилка: ");
                   }
                   else
                   {
                       printf("Результат на мові C++ X = %hd\n", x_c);
                   }
                }
        }

        else if (a < b)
        {
                if (b == 0)
                {
                     printf("Помилка: ");
                }

                else if ((a - 5)<-32768 || (a - 5)>32767)
                {
                     printf("Помилка: ");
                }
```

```
        else
        {
             x_c = (a - 5) / b;
             printf("Результат на мові C++ X = %hd\n", x_c);
        }
}

else
{
     x_c = -125;
     printf("Результат на мові C++ X = %hd\n", x_c);
}

__asm
{
        mov ax, a       // <ax> = a
        mov bx, b       // <bx> = b
        cmp ax, bx      // порівнюємо значення регістрів <ax> та
                        <bx>

        jg mark1        // a > b
        je mark2        // a = b
        jl mark3        // a < b

        mark1:
                cmp ax, 0       // порівнюємо значення <bx> з
                                нулем
                je error1       // помилка "ділення на 0"
                xchg ax, bx     // міняємо місцями значення
                                регістрів <ax> і <bx>
                mov cx, 52      // <cx> = 52
                imul cx         // <eax> = 52*b
                idiv bx         // <ax> = (52*b)/a
                add ax, b       // <ax> = (52*b)/a+b
                jo error2               // помилка "переповнення"
                mov x_asm, ax   // x = (52*b)/a+b
                jmp exit1       // переходимо до мітки exit1

        mark2:
                mov x_asm, -125 // x_asm = -125
                jmp exit1       // переходимо до мітки exit1

        mark3:
                cmp bx, 0       // порівнюємо значення <bx> з
                                нулем
                je error3       // помилка "ділення на 0"
                sub ax, 5       // <ax> = a-5
                jo error4       // помилка "переповнення"
                cwd             // <dx:ax> = a-5
```

```
                idiv bx         // <ax> = (a-5)/b
                mov x_asm, ax   // x_asm = (a-5)/b
                jmp exit1       // переходимо до мітки exit1


        error1:
                mov er, 1
                jmp exit1

        error2:
                mov over, 1
                jmp exit1

        error3 :
                mov er, 1
                jmp exit1

        error4:
                mov over, 1
                jmp exit1

        exit1:
    }

    if (er == 1)
    {
       printf("ділення на 0!\n");
    }

    else if (over == 1)
    {
       printf("переповнення!\n");
    }

    else if (er == 0 && over == 0)
    {
       printf("Результат на мові Assembler X = %hd\n", x_asm);
    }

    system ("Pause");
    return 0;
}
```

Перевіримо виникнення помилки при діленні на 0(рис. 6.7)

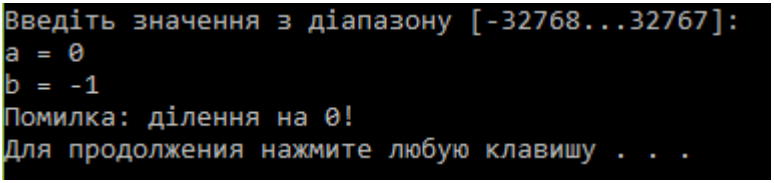

Рисунок 6.7 – Виникнення помилки при діленні на 0

Підберемо значення для перевірки виникнення переповнення(рис. 6.8)

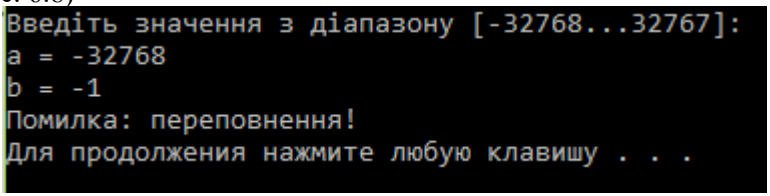

Рисунок 6.8 – Виникнення переповнення

Завдання на лабораторну роботу

1. Написати програму для обчислення заданого умовного цілочисельного виразу (табл. 6.3) для 8-бітних даних, використовуючи команди порівняння, умовного і безумовного переходів. Результат

X – теж цілочисельний і його діапазон (формат) залежить від специфіки вирішуваного умовного виразу. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Таблиця 6.3

Дані для розрахунку

№ Варіанту	Вираз
1	$X = \begin{cases} (a-b)/a-3, & якщо \quad a>b, \\ 2, & якщо \quad a=b, \\ (a^3+1)/b, & якщо \quad a<b; \end{cases}$
2	$X = \begin{cases} a/b+10, & якщо \quad a<b, \\ -51, & якщо \quad a=b, \\ (a*b-4)/a, & якщо \quad a>b; \end{cases}$
3	$X = \begin{cases} a/b-1, & якщо \quad a<b, \\ -25, & якщо \quad a=b, \\ (b^3-5)/a, & якщо \quad a>b; \end{cases}$
4	$X = \begin{cases} (a*b-1)/a, & якщо \quad a>b, \\ 255, & якщо \quad a=b, \\ (a-5)/b, & якщо \quad a<b; \end{cases}$
5	$X = \begin{cases} a/b+31, & якщо \quad a>b, \\ -25, & якщо \quad a=b, \\ (5*b-1)/a, & якщо \quad a<b; \end{cases}$
6	$X = \begin{cases} b/a+1, & якщо \quad a<b, \\ 25, & якщо \quad a=b, \\ (a^3-5)/b, & якщо \quad a>b; \end{cases}$
7	$X = \begin{cases} a/b+1, & якщо \quad a<b, \\ -2, & якщо \quad a=b, \\ (a-b)/a, & якщо \quad a>b; \end{cases}$

8	$X=\begin{cases} b/a-1, & якщо \quad a<b, \\ -295, & якщо \quad a=b, \\ (a-235)/b, & якщо \quad a>b; \end{cases}$
9	$X=\begin{cases} a/b+1, & якщо \quad a>b, \\ a+25, & якщо \quad a=b, \\ (a*b-2)/a, & якщо \quad a<b; \end{cases}$
10	$X=\begin{cases} (a*a-b)/a, & якщо \quad a>b, \\ -a, & якщо \quad a=b, \\ (a*b-1)/b, & якщо \quad a<b; \end{cases}$
11	$X=\begin{cases} a/b-1, & якщо \quad a<b, \\ 25-a, & якщо \quad a=b, \\ (b-5)/a, & якщо \quad a>b; \end{cases}$
12	$X=\begin{cases} a/b+2, & якщо \quad a>b, \\ 8, & якщо \quad a=b, \\ (b-9)/a, & якщо \quad a<b; \end{cases}$
13	$X=\begin{cases} a/b+1, & якщо \quad a<b, \\ -71, & якщо \quad a=b, \\ (a-b)/a, & якщо \quad a>b; \end{cases}$
14	$X=\begin{cases} -5+b/a, & якщо \quad a>b, \\ 45, & якщо \quad a=b, \\ (3*a-6)/b, & якщо \quad a<b; \end{cases}$
15	$X=\begin{cases} a/b-4, & якщо \quad a<b, \\ -55, & якщо \quad a=b, \\ (b-5)/a, & якщо \quad a>b; \end{cases}$
16	$X=\begin{cases} a/b+11, & якщо \quad a>b, \\ -11, & якщо \quad a=b, \\ (3*b-9)/a, & якщо \quad a<b; \end{cases}$

17	$X=\begin{cases}2*a/b+1, & якщо \quad a>b,\\ -5, & якщо \quad a=b,\\ \left(a^3-9\right)/a, & якщо \quad a<b;\end{cases}$
18	$X=\begin{cases}b/a+1, & якщо \quad a>b,\\ -20, & якщо \quad a=b,\\ (a-45)/b, & якщо \quad a<b;\end{cases}$
19	$X=\begin{cases}a/b+2, & якщо \quad a>b,\\ -100, & якщо \quad a=b,\\ (b-100)/a, & якщо \quad a<b;\end{cases}$
20	$X=\begin{cases}a/b+1, & якщо \quad a>b,\\ -100, & якщо \quad a=b,\\ (a*b-9)/a, & якщо \quad a<b;\end{cases}$
21	$X=\begin{cases}a/b-a, & якщо \quad a<b,\\ -b & якщо \quad a=b,\\ (a*b-8)/a, & якщо \quad a>b;\end{cases}$
22	$X=\begin{cases}b^2/a+1, & якщо \quad a>b,\\ 2*a, & якщо \quad a=b,\\ (a-10)/b, & якщо \quad a<b;\end{cases}$
23	$X=\begin{cases}b/a+1, & якщо \quad a>b,\\ -271, & якщо \quad a=b,\\ (a-b)/b, & якщо \quad a<b;\end{cases}$
24	$X=\begin{cases}b/a-5, & якщо \quad a>b,\\ -195, & якщо \quad a=b,\\ (a-6)/b, & якщо \quad a<b;\end{cases}$
25	$X=\begin{cases}a/b-4, & якщо \quad a<b,\\ -155, & якщо \quad a=b,\\ \left(b^3-5\right)/a, & якщо \quad a>b;\end{cases}$

26	$X = \begin{cases} b/a - 141, & якщо \quad a < b, \\ -131, & якщо \quad a = b, \\ (3*a - 9)/b, & якщо \quad a > b; \end{cases}$
27	$X = \begin{cases} a/4 - b, & якщо \quad a > b, \\ -57, & якщо \quad a = b, \\ (a^3 - 5)/b, & якщо \quad a < b; \end{cases}$
28	$X = \begin{cases} a/b + 1, & якщо \quad a > b, \\ -25, & якщо \quad a = b, \\ (b - 45)/a, & якщо \quad a < b; \end{cases}$
29	$X = \begin{cases} b/a - 2, & якщо \quad a > b, \\ -140, & якщо \quad a = b, \\ (a - 100)/b, & якщо \quad a < b; \end{cases}$
30	$X = \begin{cases} b/a - 1, & якщо \quad a > b, \\ -120, & якщо \quad a = b, \\ (a*b - 9)/b, & якщо \quad a < b; \end{cases}$

2. Написати програму для обчислення заданого умовного цілочисельного виразу (табл. 6.3) для 16-бітних даних, використовуючи команди порівняння, умовного і безумовного переходів. Результат X – теж цілочисельний і його діапазон (формат) залежить від специфіки вирішуваного умовного виразу. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

3. Написати програму для обчислення заданого умовного цілочисельного виразу (табл. 6.3) для 32-бітних даних, використовуючи команди порівняння, умовного і безумовного переходів. Результат X – теж цілочисельний і його діапазон (формат) залежить від специфіки вирішуваного умовного виразу. Провести тестові перевірки, відмітити нормальні та аномальні результати. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні.

Контрольні питання

1. Команда порівняння CMP. Призначення та синтаксис.
2. Команди умовного переходу для будь-яких чисел.
3. Команди умовного переходу для чисел без знаку. Призначення та синтаксис.
4. Команди умовного переходу для чисел зі знаком. Призначення та синтаксис.
5. Команди умовного переходу для ознаки ZF. Призначення та синтаксис.
6. Команди умовного переходу для ознаки CF. Призначення та синтаксис.
7. Команди умовного переходу для ознаки SF. Призначення та синтаксис.
8. Команди умовного переходу для ознаки OF. Призначення та синтаксис.
9. Команди умовного переходу для ознаки PF. Призначення та синтаксис.
10. Команда умовного переходу JCXZ. Призначення та синтаксис.

Лабораторна робота № 7
ОРГАНІЗАЦІЯ ЦИКЛІВ І РОБОТА З ЦІЛОЧИС-ЛЕНИМИ МАСИВАМИ

Мета заняття: ознайомитися з основними командами мови Assembler для організації циклів і роботи з цілочисельними масивами; набути практичних навичок в написанні програм з використанням циклів та масивів на мові Assembler.

Теоретичні відомості

Команди управління циклами LOOPx

Команди цього виду організують циклічні обчислення (LOOP - цикл), використовуючи регістр лічильника CX по своєму прямому призначенню. В регістр CX має бути попередньо занесена кількість повторень циклу. Ці команди реалізують класичний цикл з лічильником з постумовою.

1. Команда LOOP – перехід по лічильнику

Для організації цикла призначена команда LOOP. У цієї команди один операнд – ім'я мітки, на яку здійснюється перехід. В якості лічильника цикла використовується регістр CX. Команда LOOP виконує декремент CX, а потім перевіряє його значення. Якщо вміст CX не дорівнює нулю, то виконується перехід на мітку, інакше управління переходить до наступної після LOOP команди.

Вміст CX інтерпретується командою як число без знака. В CX потрібно поміщати число, яке дорівнює необхідній кількості повторень циклу. Зрозуміло, що максимально може бути 65535 повторень. Ще одне обмеження пов'язане з дальністю переходу. Мітка має знаходитись в діапазоні 127…+128 байт від команди LOOP (якщо це не так, FASM повідомить про помилку).

Синтаксис команди:

LOOP коротка_мітка

Логіка роботи команди:

<CX> = <Counter>

Short_label: Виконання тіла циклу

<CX> = <CX> - 1

if (<CX> < > 0) goto short_label

Аналог реалізації команди LOOP на Асемблері:

MOV CX, Counter

```
short_label:
    ; Виконання тіла циклу
    ;Перевірка умови ПРОДОВЖЕННЯ цикла
    DEC CX
    CMP CX, 0
    JNE short_label
```

Команда LOOP зменшує вміст регістра CX на 1. Потім передає управління мітці short_label, якщо вміст CX не дорівнює 0. Оскільки умова виходу із циклу перевіряється в кінці, при значенні Counter=0 цикл все одно виконається.

Щоб цього уникнути, зазвичай до початку циклу перевіряють вміст регістра CX на нуль. Таким чином, стандартна послідовність команд для організації цикла з лічильником має наступний вигляд:

```
MOV CX, Counter
JCXE ExitCicle; якщо <CX> = 0, цикл обійти
short_label:
;Виконання тіла циклу
LOOP short_label
```

Іноді потрібно організувати вкладений цикл, тобто цикл всередині іншого циклу. В цьому випадку необхідно зберегти значення CX перед початком вкладеного циклу і відновити після його закінчення (перед командою LOOP зовнішнього циклу). Зберегти значення можна в інший регістр, в тимчасову змінну або в стек.

2. Команда LOOPE (LOOPZ) перехід по лічильнику і якщо дорівнює

Дана команда має два рівнозначних мнемонічних імені (if Equal – якщо Дорівнює або if Zero – якщо Нуль). Мнемоніка команди LOOPZ передбачає знання того факту, що якщо два операнди однакові, прапорець нуля ZF=1 (встановлений).

Синтаксис команди:

LOOPE коротка_мітка
LOOPZ коротка_мітка

Логіка роботи команди:

<CX> = <Counter>
Short_label: Виконання тіла циклу
<CX> = <CX> - 1
if (<CX> < > 0 and <ZF>=1) goto short_label

Все, що було сказане для команди LOOP, виконується і для команди LOOPE (LOOPZ), додається тільки перевірка прапорця ZF. Застосовується дана команда у випадку, якщо потрібно просто вийти з циклу, як тільки знаходиться перший елемент, відмінний від заданої величини.

3. Команда LOOPNE (LOOPNZ) перехід по лічильнику і якщо не дорівнює

Дана команда також має два рівнозначних мнемонічних імені (if Not Equal – якщо Не дорівнює або if Not Zero – якщо Не нуль). На відміну від попередньої команди перевіряється, чи скинутий прапорець нуля ZF=0.

Синтаксис команди:

LOOPNE коротка_мітка
LOOPNZ коротка_мітка

Логіка роботи команди:

<CX> = <Counter>

Short_label: Виконання тіла циклу

<CX> = <CX> - 1

if (<CX> < > 0 and <ZF>=0) goto short_label

Все, що було сказане для попередньої команди, виконується і для команди LOOPNE (LOOPNZ). Застосовується дана команда у випадку, якщо потрібно просто вийти з циклу, як тільки знаходиться перший елемент, рівний заданій величині.

Основні принципи організації і обробки масивів

Масив – структурований тип даних, який складається із деякої кількості елементів одного типу. Впорядкованість масива визначається набором цілих чисел, які називаються індексами. Індекси зв'язуються з кожним елементом масиву і однозначно конкретизують його розміщення серед інших елементів масиву. Локалізація конкретного елемента масиву – основна задача при розробці будь-яких алгоритмів, що працюють з масивами. Її складність напряму залежить від розмірності масиву.

1. Одномірні масиви

Одномірні масиви мають найпростіше представлення. Структура зберігання, яка їм відповідає – вектор. Вона однозначна, і пред-

ставляє собою послідовне розміщення елементів в пам'яті. Щоб локалізувати потрібний елемент одномірного масиву, необхідно знати його індекс. Так як асемблер не має засобів для для роботи з масивом як з структурою даних, то для доступу до елементу масиву, необхідно визначити його адресу.

Нехай два масиви і вказівники на них на мові C/C++ описані наступним чином:

int ArrI[10], *PtI; PtI = ArrI;

float ArrF [5], *PtF; PtF = ArrF;

Зобразимо у вигляді таблиць основні характеристики цих масивів.

Таблиця 7.1

Основні характеристики цілочисельного масиву int ArrI[10] на мові C/C++

Байти в ОЗП	1	2	3	4	…	17	18	19	20
Елементи масиву	ArrI[0]		ArrI[1]		…	ArrI[8]		ArrI[9]	
Вказівники	ArrI		ArrI+1		…			ArrI+9	
	PtI		PtI+1		…	PtI+8		PtI+9	

Таблиця 7.2

Основні характеристики дійсного масиву float ArrF[5]на мові C/C++

Байти в ОЗП	1	2	3	4	5	6	7	8	…	17	18	19	20
Елементи масиву	ArrF[0]				ArrF[1]				…	ArrF[4]			
Вказівники	ArrF				ArrF+1				…	ArrF+4			
	PtF				PtF+1				…	PtF+4			

PtF + 2 означає те ж саме, що і &ArrF[2] і ArrF+2 – це одна і та ж адреса масиву ArrF[2].

Для того, щоб обробляти масив в Асемблері, потрібно знати, де він зберігається, і довжину його елемента. Ім'я масиву в Асемблері є також і його початковою адресою.

Режим адресації з індексацією вигляду *ім'я_масиву [регістр_індексу]* дозволяє обробляти кожний елемент масиву. В якос-

ті *регістру_індексу* можна використовувати будь-який допустимий для непрямої адресації регістр, наприклад, регістр BX.

Таблиця 7.3

Основні характеристики дійсного масиву float ArrF[5] в Асемблері

Байти в ОЗП	1	2	3	4	5	6	7	8	…	17	18	19	20
Індекс-змінна	0				1				…	4			
Елементи масиву	ArrF[0]				ArrF[1]				…	ArrF[4]			
Адреси (зміщення)	ArrF	ArrF+1	ArrF+2	ArrF+3	…				…	ArrF+16	ArrF+17	ArrF+18	ArrF+19
Індекс-регістр	BX_0				BX_0+4				…	BX_0+4*i			

В загальному випадку, якщо взяти в якості індекс-регістра регістр BX, доступ до будь-якого елемента одномірного масиву Array [i] довжини Larray підпорядковується в Асемблері наступній закономірності:

Array[i] → Array + BX_i = Array[BX_i], де

$BX_i = BX_0$ + Larray*I = BX_{i-1} + Larray;

BX_0 = 0; i=0,…,n-1; n – довжина масиву Array.

2. Двомірні масиви

Для двомірних масивів ідея та ж сама, тільки необхідно визначитися, як такий масив буде розміщуватися в оперативній пам'яті: по рядкам чи по стовбцям. Відповідно і індекс-регістрів також буде два. А також два цикли – внутрішній і зовнішній.

Наприклад, нехай є якась матриця Matr[M][N] і в пам'яті вона розміщується по рядкам: спочатку N елементів першого рядка, потім N елементів другого рядка і т.д. до рядка M. Довжину елемента цієї матриці також позначим через Larray.

Тоді адреса елемента Matr[i,j] буде дорівнювати Matr+N*i*Larray+j, де i=0,…, M-1; j=0,…,N-1. Виділимо в Асемблері для зберігання величини N*i*Larray регістр BX, а для j – регістр SI (або DI). Тоді Matr[BX] буде означати початкову адресу рядка i, а Matr[BX][SI] (Matr[BX+SI] або Matr+[BX+SI] – ці три записи рівнозначні) – адреса елемента j в цьому рядку, тобто Matr[i,j] → Matr[BX][SI].

Таблиця 7.4

Основні характеристики цілочисельної матриці A[10][5] в Асемблері

Байти	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	97	98	99	100
Стовбці (j)	0		1		2		3		4		0		1		…	3		4	
Індекс-регістр SI	0		+2		+4		+6		+8		0		+2		…	+6		+8	
Рядки (i)	0										1				…	9			
Індекс-регістр (BX)	0										+(10*1)=+10				…	+(10*9)=+90			

Приклад 1: Написати програму для обробки одномірного масиву для початкових даних в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Знайти, скільки елементів масиву A={a[i]} задовольняють умові: c<=a[i]<=d. Тип даних INTEGER.

Лістинг програми:

```
include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#define N 10
int _tmain(int argc, _TCHAR* argv[])
{
  short int a[N];
  short int c = 0, d = 0;
  printf("c<=a[i]<=d\n\n");
  do{
      printf("Enter    the    values    of    the    range    [-
32768...32767]:\n");
      printf("c = "); scanf_s("%d", &c);
      printf("d = "); scanf_s("%d", &d);
```

```
      if (c >= d)
      {
       printf("c can not be greater or equal d! Enter values
again.\n\n");
      }
  } while (c >= d);
  int n = N;
  short int res = 0;
  for (int i = 0; i < N; i++)
  {
     a[i] = rand() % 10;
     printf("A[%d] = %d\n", i, a[i]);
  }
  __asm
  {
     mov ax, c  // <ax> = c
     mov bx, d  // <bx> = d
     mov ecx, n // <ecx> = n
     mov dx, 0  // <dx> = 0
     dec ecx    // зменшуємо значення в регістрі <ecx> на 1
     cycle:     // цикл cycle
     shl ecx, 1 // зсув вліво на 1 розряд
     mov si, a[ecx] // <si> = a[ecx]
     cmp si, 0      // порівнюємо значення регістра <si> з 0
     jl exit1       // якщо менше - перейти до циклу exit1
     cmp si, ax     // порівнюємо значення регістра <si> з
                    значенням в регістрі <ax>
     jl exit1       // якщо менше - перейти до циклу exit1
     cmp si, bx     // порівнюємо значення регістра <si> з
                    значенням в регістрі <bx>
     jg exit1       // якщо більше - перейти до циклу exit1
     inc dx         // збільшуємо значення в регістрі <dx> на 1
     exit1:         // цикл exit1
     shr ecx, 1     // зсув вправо на 1 розряд
     dec ecx        // зменшуємо значення в регістрі <ecx> на 1
     cmp ecx, 0     // порівнюємо значення регістра <ecx> з 0
     jnl cycle      // поки не менше - перейти до циклу cycle
     mov res, dx    // res = <dx>
  }
  if (res > 32767 || res < -32768)
  {
     printf("Overflow!\n");
  }
  else
  {
```

```
    printf("Result = %d\n", res);
  }
    _getch();
    return 0;
}
```

Проведемо тестову перевірку роботи програми(рис.7.1)

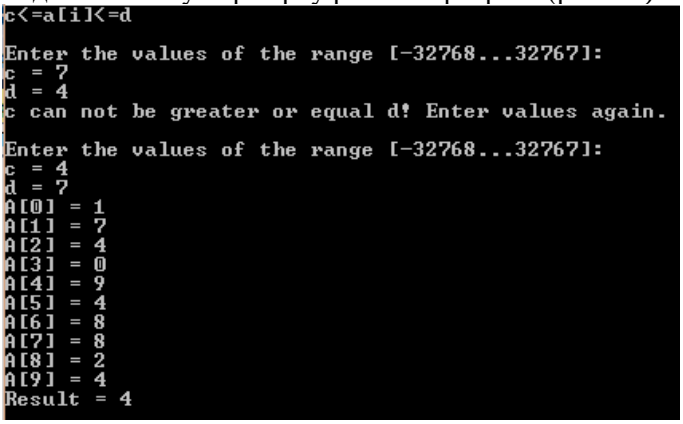

Рисунок 7.1 – Перевірка роботи програми

Значення регістрів при покроковому виконанні

Команда	Значення регістра				EFLAGS/FLAGS (CF, OF)
	ax	bx	ecx	dx	
mov ax, c	4	н/в	н/в	н/в	–
mov bx, d	н/в	7	н/в	н/в	–
mov ecx, n	н/в	н/в	10	н/в	–
mov dx, 0	н/в	н/в	н/в	0	–
dec ecx	н/в	н/в	9	н/в	–
cycle:	Мітка циклу cycle(код буде виконуватись в цій мітці, якщо умова – істина)				
shl ecx, 1	Зсув біта операнда вліво на 1 розряд				
mov si, a[ecx]	Заносимо в регістр <si> елемент масиву з відповідним індексом				
cmp si, 0	Крок масиву порівнюється з 0				
jl exit1	Якщо менше – перейти до мітки циклу exit1				
cmp si, ax	Порівнюємо елемент масиву з нижньою границею – c				
jl exit1	Якщо менше – перейти до мітки циклу exit1				
cmp si, bx	Порівнюємо елемент масиву з верхньою границею – d				

jg exit1	Якщо більше – перейти до мітки циклу exit1				
inc dx	Збільшуємо значення в регістрі <dx> на 1				
exit1:	Мітка циклу exit1(код буде виконуватись в цій мітці, якщо умова – істина)				
shr ecx, 1	Зсув біта операнда вправо на 1 розряд				
dec ecx	Зменшуємо значення в регістрі <ecx> на 1				
cmp ecx, 0	Порівнюємо значення регістра <ecx> з 0				
jnl cycle	Перейти на мітку циклу, якщо умова є неві				
mov res, dx	н/в	н/в	н/в	4	–

Приклад 2: Написати програму для обробки двовимірного масиву для початкових даних в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Знайти, скільки елементів масиву A={a[i][j]} задовольняють умові: c<=a[i][j]<=d. Тип даних INTEGER.

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#define N 5

int _tmain(int argc, _TCHAR* argv[])
{
   short int a[N][N];
   short int c = 0, d = 0;
   do{
        printf("c<=a[i]<=d\n\n");
        printf("Enter the values c and d:\n");
        printf("c = "); scanf_s("%d", &c);
        printf("d = "); scanf_s("%d", &d);
        printf("\n");
        if (c >= d)
        {
```

```
        printf("c can not be greater or equal d! Enter values
again.\n");
         }
   } while (c >= d);
   int n = N * N; // загальна кількість елементів масиву
   short int res = 0;
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
       a[i][j] = rand() % 10;
       printf("%d ", a[i][j]);
      }
   printf("\n");
   }
   __asm
   {
      mov bx, c  // <bx> = c
      mov dx, d  // <dx> = d
      mov ecx, n // <ecx> = n
      lea si, a  // завантаження зміщення масиву в регістр
                     SI(замінює shr ecx, 1 та shl ecx, 1)
      cycle:     // цикл cycle
       lodsw      // завантажуємо слово з пам'яті, на який
                        вказує регістр SI
       cmp ax, bx // порівнюємо з нижньою границею - c
       jl next    // якщо менше - перейти до циклу next
       cmp ax, dx // порівнюємо з верхньою границею - d
       jg next    // якщо більше - перейти до циклу next
       inc res    // збільшити результат на 1
      next:            // цикл next
       dec ecx    // зменшуємо к-ть чисел на 1
       cmp ecx, 0 // порівнюємо к-ть чисел з 0
       jnl cycle  // якщо не менше - перейти до циклу
                        cycle
   }
   if (res > 32767 || res < -32768)
   {
      printf("Overflow!\n");
   }
   else
   {
      printf("\n");
      printf("Result = %d\n", res);
   }
```

```
    _getch();
    return 0;
}
```

Проведемо тестову перевірку роботи програми(рис.7.2)

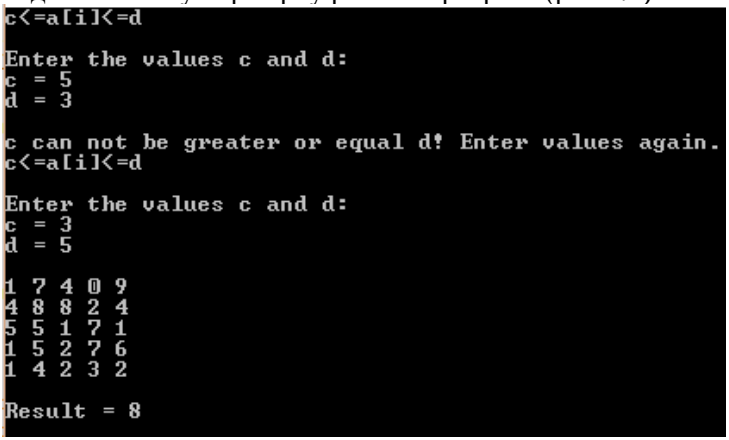

Рисунок 7.2 – Перевірка роботи програми

Значення регістрів при покроковому виконанні

Команда	**EFLAGS/FLAGS (CF, OF)**			
	bx	**dx**	**ecx**	–
mov bx, c	3	н/в	н/в	–
mov dx, d	н/в	5	н/в	–
mov ecx, n	н/в	н/в	25	–
lea si, a	Беремо в регістр <si> адресу першого елемента масиву			
cycle:	Мітка циклу cycle(код буде виконуватись в цій мітці, якщо умова – істина)			
lodsw	Завантажуємо слово з пам'яті, на який вказує регістр SI			
cmp ax, bx	Порівнюємо елемент масиву з нижньою границею – c			
jl next	Якщо менше – перейти до мітки циклу next			
cmp ax, dx	Порівнюємо елемент масиву з верхньою границею – d			
jg next	Якщо більше – перейти до мітки циклу next			
inc res	Збільшуємо результат res на 1			
next:	Мітка циклу next(код буде виконуватись в цій мітці, якщо умова – істина)			
dec ecx	Зменшуємо значення в регістрі <ecx> на 1			
cmp ecx, 0	Порівнюємо значення регістра <ecx> з 0			
jnl cycle	Поки індекс елементу масиву не менше 0 – перейти до циклу cycle			

Завдання на лабораторну роботу

1. Написати програму для обробки одномірного масиву для початкових даних(табл.7.5) в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Таблиця 7.5

Дані для розрахунку

№ Варіанту	Вираз
1	Знайти, скільки елементів масиву **A={a[i]}** задовольняють умові: c<=a[i]<=d. Тип даних **BYTE.**
2	Знайти, скільки елементів масиву **A={a[i]}** задовольняють умові: c<=a[i]<=d. Тип даних **SHORTINT.**
3	Знайти, скільки елементів масиву **A={a[i]}** задовольняють умові: c<=a[i]<=d. Тип даних **WORD**.
4	Знайти суму перших k чисел массиву **A={a[i]}**. Тип даних **INTEGER**.
5	Знайти добуток елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **BYTE**.
6	Знайти добуток елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **SHORTINT**.
7	Знайти добуток елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **WORD**.
8	Знайти добуток елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **INTEGER**.
9	Знайти кількість від'ємних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d**.** Тип даних **SHORTINT**.
10	Знайти кількість від'ємних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d**.** Тип даних **INTEGER**.
11	Знайти кількість додатних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d**.** Тип даних **SHORTINT**.
12	Знайти кількість додатних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d**.** Тип даних **INTEGER.**
13	Знайти суму квадратів всіх додатних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **SHORTINT.**
14	Знайти суму квадратів всіх додатних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **INTEGER.**
15	Знайти суму квадратів всіх від'ємних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **SHORTINT.**
16	Знайти суму квадратів всіх від'ємних елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **INTEGER.**

Продовження табл. 7.5

Дані для розрахунку

№ Варіанту	Вираз
17	Знайти скільки додатних, від'ємних і нульових елементів у масиві **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **SHORTINT.**
18	Знайти скільки додатних, від'ємних і нульових елементів у масиві **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **INTEGER.**
19	Знайти скільки додатних і нульових елементів у масиві **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **BYTE.**
20	Знайти скільки додатних і нульових елементів у масиві **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **WORD.**
21	Знайти суму елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **BYTE**.
22	Знайти суму елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **WORD**.
23	Знайти суму елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **SHORTINT**.
24	Знайти суму елементів масиву **A={a[i]}**, які задовольняють умові c<=a[i]<=d. Тип даних **INTEGER**.
25	Знайти кількість нульових елементів масиву **A={a[i]}**. Тип даних **BYTE**.
26	Знайти кількість нульових елементів масиву **A={a[i]}**. Тип даних **WORD**.
27	Знайти кількість ненульових елементів масиву **A={a[i]}**. Тип даних **SHORTINT**.
28	Знайти кількість ненульових елементів масиву **A={a[i]}**. Тип даних **WORD**.
29	Знайти суму перших k чисел массиву **A={a[i]}**. Тип даних **BYTE.**
30	Знайти суму перших k чисел массиву **A={a[i]}**. Тип даних **WORD.**

2. Написати програму для обробки двовимірного масиву для початкових даних(табл.7.6) в знаковому форматі. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Таблиця 7.6

Дані для розрахунку

№ Варіанту	Вираз
1	Знайти, скільки елементів масиву **A={a[i][j]}** задовольняють умові: c<=a[i][j]<=d. Тип даних **BYTE.**

Продовження табл. 7.6

Дані для розрахунку

№	Вираз

Варіанту	
2	Знайти, скільки елементів масиву **A={a[i][j]}** задовольняють умові: c<=a[i][j]<=d. Тип даних **SHORTINT.**
3	Знайти, скільки елементів масиву **A={a[i][j]}** задовольняють умові: c<=a[i][j]<=d. Тип даних **WORD**.
4	Знайти суму перших k чисел массиву **A={a[i][j]}**. Тип даних **INTEGER**.
5	Знайти добуток елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **BYTE**.
6	Знайти добуток елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **SHORTINT**.
7	Знайти добуток елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **WORD**.
8	Знайти добуток елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **INTEGER**.
9	Знайти кількість від'ємних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d**.** Тип даних **SHORTINT**.
10	Знайти кількість від'ємних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d**.** Тип даних **INTEGER**.
11	Знайти кількість додатних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d**.** Тип даних **SHORTINT**.
12	Знайти кількість додатних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d**.** Тип даних **INTEGER.**
13	Знайти суму квадратів всіх додатних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **SHORTINT.**
14	Знайти суму квадратів всіх додатних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **INTEGER.**
15	Знайти суму квадратів всіх від'ємних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **SHORTINT.**
16	Знайти суму квадратів всіх від'ємних елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **INTEGER.**
17	Знайти скільки додатних, від'ємних і нульових елементів у масиві **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **SHORTINT.**
18	Знайти скільки додатних, від'ємних і нульових елементів у масиві **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **INTEGER.**
19	Знайти скільки додатних і нульових елементів у масиві **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **BYTE.**
20	Знайти скільки додатних і нульових елементів у масиві **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **WORD.**

Продовження табл. 7.6

Дані для розрахунку

№ Варіанту	Вираз
21	Знайти суму елементів масиву **A={a[i][j]}**, які задовольняють умові

	c<=a[i][j]<=d. Тип даних **BYTE.**
22	Знайти суму елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **WORD**.
23	Знайти суму елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **SHORTINT**.
24	Знайти суму елементів масиву **A={a[i][j]}**, які задовольняють умові c<=a[i][j]<=d. Тип даних **INTEGER**.
25	Знайти суму кількість нульових елементів масиву **A={a[i][j]}**. Тип даних **BYTE**.
26	Знайти суму кількість нульових елементів масиву **A={a[i][j]}**. Тип даних **WORD**.
27	Знайти суму кількість ненульових елементів масиву **A={a[i][j]}**. Тип даних **SHORTINT**.
28	Знайти суму кількість ненульових елементів масиву **A={a[i][j]}**. Тип даних **WORD**.
29	Знайти суму перших k чисел массиву **A={a[i][j]}**. Тип даних **BYTE.**
30	Знайти суму перших k чисел массиву **A={a[i][j]}**. Тип даних **WORD.**

Контрольні питання

1. Організація циклів на мові Ассемблер з використанням команд умовних та безумовних переходів.
2. Організація циклів на мові Ассемблер з використанням команд LOOPx.
3. Команда організації циклів LOOP. Призначення та синтаксис.
4. Особливості застосуванняк команди JCXZ при організації циклів на мові Ассемблер.
5. Типова послідовність команд для організації циклу в мові Ассемблер.
6. Команда організації циклів LOOPE. Призначення та синтаксис.
7. Команда організації циклів LOOPZ. Призначення та синтаксис.
8. Команди управління циклом LOOPNE. Призначення та синтаксис.
9. Команди управління циклом LOOPNZ. Призначення та синтаксис.
10. Написати модулі на мові Ассемблер для виконання сумування чисел від 1 до N. Організацію циклів у відповідних модулях виконати з використанням команд умовних та безумовних переходів, а також команди організації циклів LOOP.

Лабораторна робота № 8
ОБЧИСЛЕННЯ АРИФМЕТИЧНИХ ВИРАЗІВ І ТРАНСЦЕНДЕНТНИХ ФУНКЦІЙ (СПІВПРОЦЕСОР ix87)

Мета заняття: ознайомитися з основними командами мови Assembler для обчислення арифметичних виразів і трансцендентних функцій; набути практичних навичок в написанні програм, які обчислюють заданий змішаний арифметичний вираз, використовуючи арифметичні операції співпроцесора на мові Assembler.

Теоретичні відомості

Математичний співпроцесор (FPU – Floating Point Unit) – спеціальний пристрій, який слугує для обробки дійсних чисел (чисел з плаваючою комою).

З моменту свого виникнення співпроцесор розширював обчислювальні можливості основного процесора i8086 (i80286, i82386, i80486) і спочатку був виконаний у вигляді окремої мікросхеми i8087 (i80287, i80387, i80487).Його присутність в перших моделях процесора була не обов'язковою. Починаючи з сімейства процесорів i486DX співпроцесор став складовою частиною основного процесора.

Сучасний співпроцесор забезпечує повну підтримку стандартів IEEE-754 і IEEE-854 по представленню і обробці чисел з плаваючою комою. Він може виконувати трансцендентні операції (обрахування тригонометричних функцій, логарифмів і т.д.) з більшою точністю.

Типи даних

Співпроцесор може виконувати операції з сімома звичайними типами даних, а також із спеціальними числами.

Звичайні дані можуть бути дійсними або цілими числами із знаком. Наявність цілочисельних даних дозволяє виконувати так звані змішані обрахування, коли в якості операндів використовуються і цілі, і дійсні числа.

Співпроцесор виконує всі обрахування в 80-бітному розширеному дійсному форматі, а 16-, 32- і 64-бітні дані використовуються

для обміну даними і можуть застосовуватись в командах співпроцесора як операнди.

Таблиця 8.1

Типи звичайних даних співпроцесора

Тип даних	Довжина (біт)	К-сть значимих цифр	Діапазон представлення
Ціле слово	16	4-5	-32768…32767
Коротке ціле	32	9-10	-2147483648 … -2147483647
Довге ціле	64	18-19	-9223372036854775808 … 9223372036854775807
Упаковане двійково-десяткове	80	18	-999999999999999999 … 999999999999999999
Коротке дійсне	32	7-8	1.175494351e-38 … 3.402823466e+38
Довге дійсне	64	15-16	2.2250738585072014e-308... 1.7976931348623158e+308
Розширене дійсне	80	19-20	3.3621031431120935063e-4932… 1.189731495357231765e+4932

Крім звичайних чисел, специфікація стандарту IEEE передбачає декілька спеціальних форматів, які можуть виникнути в результаті виконання математичних операцій співпроцесора і над якими можна виконувати окремі операції.

1. **Додатний нуль –** всі біти числа скинуті до нуля.

2. **Від'ємний нуль** – знаковий біт дорівнює 1, всі інші біти числа скинуті до нуля.

3. **Додатня нескінченність –** знаковий біт дорівнює 0, всі біти експоненти установлені в 1, а біти мантиси скинуті до нуля.

4. **Від'ємна нескінченність -** знаковий біт дорівнює 1, всі біти експоненти установлені в 1, а біти мантиси скинуті до 0.

5. **Денормалізовані числа –** всі біти експоненти скинуті до 0. Ці числа дозволяють представляти дуже маленькі числа при обрахунках з розширеною точністю.

6. **Невизначеність –** знаковий біт дорівнює1, перший біт мартиси дорівнює 1 (для 80-розрядних чисел перші два біта дорівнюють

11), інші біти мантиси скинуті до нуля, всі біти експоненти встановленні в 1.

7. **Не-число типу SNAN (сигнальне) –** перший біт мантиси дорівнює 0 (для 80-розрядних чисел перші два біти дорівнюють 10), інші біти мантиси можуть містити 0 або 1, всі біти експоненти встановлені в 1.

8. **Не-число типу QNAN (тихе) –** перший біт мантиси дорівнює 1 (для 80-розрядних чисел перші два біти дорівнюють 11), інші біти мантиси можуть містити 0 або 1, всі біти експоненти встановлені в 1.

9. **Непідтримуване число –** всі інші ситуації.

Регістри

У співпроцесора є 8 регістрів для зберігання даних R0-R7 і 5 допоміжних регістрів, які складають середовище співпроцесора.

Регістри даних співпроцесора R0-R7 мають довжину 80 біт (тобто 16-розрядних слів) і розглядаються як круговий стек, вершина якого (TOP) називається ST або ST(0) і є плаваючою.

Принцип роботи з круговим стеком співпроцесора аналогічний звичайному калькулятору. Будь-яка команда завантаження даних співпроцесора автоматично переміщує вершину стеку співпроцесора: TOP = TOP + 1.

В таблиці 8.2 показана гіпотетична ситуація, коли в результаті виконання якоїсь команди вершиною стека став регістр R6. Інші регістри розподіляються по колу: R7-ST(1), R0-ST(2), …, R5-ST(7). Це ї є їх поточні імена ST(i), i=1,…,7 на момент виконання данної команди співпроцесора.

Якщо в цих регістрах є дані, то вони можуть слугувати операндами в командах співпроцесора. Звертатися ж напряму до регістрів R0-R7 не можна.

Таблиця 8.2

Регістри співпроцесора і поняття вершини стеку співпроцесора

Назва регістрів	**Біти**															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CWR	Регістр управління (**Control Word Register**)															

Продовження табл.8.2

Регістри співпроцесора і поняття вершини стеку співпроцесора

SWR	Регістр стану співпроцесора **(Status Word Register)**
TWR	Слово ознак-тегів **(Tags Word Register)**
FIP (32 біти)	Регістр вказівника останньої виконаної команди
	(Floating Instruction Pointer)
FDP (32)	Регістр, де зберігається адреса операнда останньої виконаної команди
	(Floating Data Pointer)
R0 – ST (2) (80 біт)	
	Розширене дійсне чи будь-яке інше доступне дане співпроцесора
R1 – ST (3) (80 біт)	
	Розширене дійсне чи будь-яке інше доступне дане співпроцесора
……	
R6 – (TOP) (80 біт)	
	Розширене дійсне чи будь-яке інше доступне дане співпроцесора
R7 – ST (1) (80 біт)	
	Розширене дійсне чи будь-яке інше доступне дане співпроцесора

Регістр стану співпроцесора сигналізує співпроцесору про його стан і про те, як виконалась та чи інша арифметична команда. Таким чином, цей регістр виконує функції, аналогічні функціям регістра прапорців процесора FLAGS. За бітами регістра стану співпроцесора закріплеі відповідні імена, які полегшують розуміння їх призначення.

Таблиця 8.3

Регістр стану співпроцесора SWR

Регістр (слово) стану співпроцесора (SWR)															
Старший байт								Молодший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP			C2	C1	C0	ES	SE	PE	UE	OE	ZE	DE	IE
Аналогія з регістром прапорців процесора (Flags)															
	ZF				PF		CF								

Регістр управління визначає для співпроцесора варіанти обробки дійсних чисел (таблиця 8.4).

Таблиця 8.4

Регістр управління співпроцесора SWR

Регістр (слово) управління співпроцесора (CWR)															
Старший байт								Молодший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			IC	RC		PC		IEM		PM	UM	OM	ZM	DM	IM

Молодші біти слова управління визначають маскування обрахованих виключень. Біти 0-5 містять індивідуальні маски для кожного із шести виключень, які розглядаються процесором при виконанні операцій з плаваючою комою. Старші біти (8-12) управляючого слова визначають параметри роботи співпроцесора.

Регістр тегів містить 8 пар бітів, які описують вміст кожного регістра даних R0-R7:

00 – число;

01 – істинний нуль;

10 – особливе число;

11 – регістр пустий.

Таблиця 8.5

Регістр (слово) тегів (TWR)

Регістр (слово) тегів (TWR)															
Старший байт								Молодший байт							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R7		R6		R5		R4		R3		R2		R1		R0	

Система команд

Система команд співпроцесора досить проста, якщо знати ключ і розуміти англійську. Для їх підключення необхідно зробити наступне:

1. Скористатись однією із директив Асемблера: .8087, .287 (.286p), .387 (.386p.), .487(.486p). Необхідно мати на увазі, що не всі команди процесора сумісні зверху вниз.

2. Зробити ініціалізацію співпроцесора за допомогою команди FINIT.

3. При компіляції використовувати додатковий ключ.

Базовий співпроцесор i8087 має 69 базових команд, які підкоряються наступним закономірностям:

1. Всі вони починаються на букву F (Floating).

2. Друга буква може бути пов'язана з форматом оброблюваних чисел:

I (Integer) – цілі числа;

B (Binary-coded decimal) – упаковане двійково-десяткове число (BCD).

Для дійсних чисел спеціальна буква не виділяється.

3. Далі іде мнемонічний код операції (МКОП), наприклад:

LD(LoaD) – завантажити дане в вершину стека ST;

ST(STore) – вивантажити дане із вершини стека ST і уже відомі нам команди **ADD, MUL, DIV** і т.д.

4. Закінчується команда може буквою **R** (Reserved – реверсна операція) і/або **P** (Pop – виштовхнути результат із стека і звільнити вершину стека ST).

Таблиця 8.6

Звичайні і реверсні операції

МКОП	Звичайна операція	Реверсна операція
SUB	Приймач=Приймач-Джерело	Приймач=Джерело-Приймач
DIV	Приймач=Приймач/Джерело	Приймач=Джерело/Приймач

Співпроцесори пізніших моделей, починаючи з i80387, порушили цю систему умовних позначень, зате збільшили функціональні можливості команд по обробці даних.

1.1 Команди пересилки даних

Всі команди цієї групи мають один операнд: або джерело, або приймач. Завантаження або вивантаження інформації відбувається в/з вершини стека ST(0).

Зазвичай вершину стека позначають через ST, і в командах цієї групи вона явно в операндах не є присутньою.

Таблиця 8.7

Команди передачі даних

Команда	Тип даних	Характеристика операнда	Приклад
Команди завантаження даних в стек			
FLD джерело	Дійсне	Змінна в ОЗП або ST(i), i=0,…7	FLD a FLD ST(5)
FBLD джерело	Типу BCD	Змінна в ОЗП	FBLD aBc
FILD джерело	Ціле	Змінна в ОЗП	FILD ka
Команди кооперування даних з стеку			
FST приймач	Дійсне	Змінна в ОЗП або ST(i), i=1,…7	FST ap FST ST(1)
FIST приймач	Ціле	Змінна в ОЗП	FIST kab
Команди вивантаження даних з стеку із звільненням вершини стеку			
FSTP приймач	Дійсне	Змінна в ОЗП або ST(i), i=1,…7	FSTP app FSTP ST(5)
FBSTP приймач	Типу BCD	Змінна в ОЗП	FBSTP ppp
FISTP приймач	Ціле	Змінна в ОЗП	FISTP kabP
Команда обміну вмісту джерела з ST(0)			
FXCH (джерело)	Якщо джерело не вказане, то вважається, що він не відповідає ST (1)	ST(i), i=1,…7	FXCH FXCH ST(4)

1.2 Команди завантаження констант

Команди цієї групи поміщають в вершину стека ST(0) константи, які часто використовуються. Починаючи з співпроцесора i80987, ці константи зберігаються в більш точному форматі. Команди операндів не мають.

Таблиця 8.8

Команди завантаження констант

Команда	Призначення
FLDI	Помістити в ST (0) число 1.0
FLDZ	Помістити в ST (0) число +0.0
FLDPI	Помістити в ST (0) число p
FLDL2E	Помістити в ST (0) число, рівне $\log_2 e$
FLDL2T	Помістити в ST (0) число, рівне $\log_2 10$
FLDLN2	Помістити в ST (0) , рівне ln 2
FLDLG2	Помістити в ST (0) , рівне lg 2

1.3 Арифметичні команди

Арифметичні команди реалізують базову арифметику співпроцесора (чотири основні арифметичні операції: +, -, *, /) і декілька спеціальних арифметичних команд.

Формат команд базової арифметики для дійсних чисел досить складний (операнди можуть бути задати в явному вигляді, а можуть і бути відсутні, тобто неявні), тому, щоб уникнути непорозумінь, в таблиці 8.9 даний явний формат.

Спеціальні арифметичні команди в своєму форматі не містять операндів, тобто всі їх операнди задані неявно.

Таблиця 8.9

Команди базової арифметики

Команда	Тип даних	Алгоритм виконання
Команди додавання		
FADD приймач, джерело	Дійсне	Приймач=приймач+джерело
FADDP приймач, джерело	Дійсне	
FIADD приймач	Ціле	
Команди звичайного віднімання		
FSUB приймач, джерело	Дійсне	Приймач=приймач-джерело
FSUBP приймач, джерело	Дійсне	
FISUB джерело	Ціле	
Команди реверсного (зворотнього) віднімання		
FSUBR приймач, джерело	Дійсне	Приймач=джерело-приймач
FSUBPR приймач, джерело	Дійсне	
FISUBR джерело	Ціле	

Продовження табл.8.9

Команди базової арифметики

Команди множення		
FMUL приймач, джерело	Дійсне	Приймач=приймач*джерело
FMULP приймач, джерело	Дійсне	
FIMUL джерело	Ціле	
Команди звичайного ділення		
FDIV приймач, джерело	Дійсне	Приймач=приймач/джерело
FDIVP приймач, джерело	Дійсне	
FIDIV джерело	Ціле	
Команди реверсного (зворотного) ділення		
FDIVR приймач, джерело	Дійсне	Приймач=джерело*приймач
FDIVRP приймач, джерело	Дійсне	
FIDIVR джерело	Ціле	

Таблиця 8.10

Спеціальні арифметичні команди

Команда	Призначення	Алгоритм виконання
FPREM	Знаходження часткової остачі від ділення шляхом послідовного віднімання 64 рази вмісту ST(1) з ST(0)	ST(0)=остача[ST(0)/ST(1)] Формується прапорець C2 регістра управління CWR: C2=0 – отримана точна остача від ділення, тобто остача < ST(1) C2=1 – точна остача не отримана
FPREM1 (i80387)	Знаходження часткової остачі від ділення в стандарті IEEE – округлення до найбільшого цілого	
FABS	Знаходження абсолютного значення	ST(0)=abs(ST(0))
FSQRT	Знаходження квадратного кореня	ST(0)=sqrt(ST(0))
FCHS	Зміна знака	ST(0)= -ST(0)
FRNDINT	Округлення до цілого	Вміст ST(0) округляється до цілого у відповідності з бітами RC регістра управління співпроцесором CWR
FXTRACT	Вивантажити експоненту і мантису числа	Число ===> ST(0) ST(0) = мантиса числа ST(1) = експонента числа
FSCALE	Команда, зворотна FXTRACT	ST(0) <=== мантиса числа ST(1) <=== експонента числа ST(0) = ST(0)*$2^{ST(1)}$

1.4 Трансценденті операції

Команди цієї групи в своєму форматі не містять операндів, тобто їх операнди задані неявно. Вони призначені для обрахування найбільш вживаних арифметичних функцій.

Таблиця 8.11

Команди трансцендентних операцій

Команда	Призначення	Алгоритм виконання
FSIN (i80387)	Обрахування синуса sin(x), де значення аргумента x задається в радіанах	x ===> ST(0); ST(0) = sin(ST(0)) $-2^{63}<=x<=2^{63}$. Якщо значення x виходить за ці межі, можна скористатися командою FPREM з дільником 2П. Інакше прапорець C2=1 і значення ST(0) не змінюється.
FCOS (i80387)	Обрахування синуса sin(x), де значення аргумента x задається в радіанах	x ===> ST(0); ST(0) = cos(ST(0)) $-2^{63}<=x<=2^{63}$. Якщо значення x виходить за ці межі, можна скористатися командою FPREM з дільником 2П. Інакше прапорець C2=1 і значення ST(0) не змінюється.
FSINCOS (i80387)	Обрахування синуса і косинуса	x ===> ST(0); ST(1) = sin(ST(0)); ST(0) = cos(ST(0)) $-2^{63}<=x<=2^{63}$. Якщо значення x виходить за ці межі, можна скористатися командою FPREM з дільником 2П. Інакше прапорець C2=1 і значення ST(0) не змінюється.
FPTAN	Обрахування часткового тангенса tg(a)=y/x, де значення аргумента a задається в радіанах	a ===> ST(0); ST(0) = x; ST(1) = y; 0<a<П/4 (i8087, i80287). $-2^{63}<=a<=2^{63}$. Якщо значення a виходить за ці межі, можна скористатися командою FPREM з дільником П. Інакше прапорець C2=1 і значення ST(0) не змінюється.
FPATAN	Обрахування арктангенса a=arctg (y/x)	ST(0) = x; ST(1) = y; a <=== ST(0); знак співпадає із знаком ST(1) 0<|y|<|x|<нескінченність. |a| < П.
F2XM1	Обрахування 2^X-1	X ===> ST(0); ST(0) = $2^X - 1$ -1<X<1, інакше результат операції не визначений.
FYL2X	Обрахування $y*\log_2(x)$	x ===> ST(0); y ===> ST(1); 0 < x < нескінченність, -нескінченність <y< +нескінченність
FYL2XP1	Обрахування $y*\log_2(c+1)$	C ===> ST(0); y ===> ST(1); $0 < \|c\| < 1-2^{0,5}/2$, -нескінченність <y< +нескінченність

1.5 Команди порівняння

Команди цієї групи виконують порівняння вмісту регістра ST(0) з джерелом і виставляють відповідні прапорці співпроцесора або процесора.

При порівнянні дійсних даних джерело може знаходитися або в регістрі ST(i), i=1,…,7, або в оперативній пам'яті. Якщо джерело в форматі команди не вказане, припускається, що воно зберігається в ST(1).

При порівнянні цілочисельних даних джерело може знаходитися тільки в оперативній пам'яті.

Таблиця 8.12

Основні команди порівняння

Команда	Призначення
FCOM джерело	Порівняння дійсних даних
FCOMP джерело	Порівняння дійсних даних і звільнення вершини стека ST(0)
FCOMPP	Порівняння дійсних даних і звільнення вершини стека ST(0) і регістра ST(1)
FUCOM джерело (i80387)	Порівняння дійсних даних без врахуваня порядків
FUCOMP джерело (i80387)	Порівняння дійсних даних без врахуваня порядків і звільнення вершини стека ST(0)
FUCOMPP (i80387)	Порівняння дійсних даних без врахуваня порядків і звільнення вершини стека ST(0) і регістра ST(1)
FICOM джерело	Порівняння цілочисельних даних
FICOMP джерело	Порівняння цілочисельних даних і звільнення вершини стека ST(0)
FCOMI ST(0), ST(i)	Порівняння даних і встановлення EFLAGS
FCOMIP ST(0), ST(i)	Порівняння даних, встановлення EFLAGS і звільнення вершини стека ST(0)
FUCOMI ST(0), ST(i)	Порівняння даних бех врахування порядків і встановлення EFLAGS
FUCOMIP ST(0), ST(i)	Порівняння даних бех врахування порядків, встановлення EFLAGS і звільнення вершини стека ST(0)
FTST	Порівняння вмісту ST(0) з нулем
FXAM	Аналіз вмісту і визначення типу числа в ST(0)

1.6 Команди управління співпроцесором

В командах цієї групи неявних операндів нема. Приймач або джерело відповідають ділянці в оперативній пам'яті необхідної довжини.

Таблиця 8.13

Команди управління співпроцесором

Команда	Призначення
FLDCW джерело	Завантаження регістрів управління CWR з джерела
FSTCW приймач	Запис вмісту слова управління CWR в приймач. Дана команда повністю еквівалентна команді WAIT FNSTCW
FNTCW приймач	Запис вмісту слова управління CWR в приймач без очікування закінчення обробки виключних ситуацій
FSTSW приймач (i80287)	Запис вмісту слова стану SWR в приймач. Дана команда повністю еквівалентна команді WAIT FNSTSW
FNSTSW приймач (i80287)	Запис вмісту слова стану SWR в приймач без очікування закінчення обробки виключних ситуацій
FSAVE приймач	Збереження повного стану FPU в приймачі. Дана команда повністю еквівалентна команді WAIT FNSAVE
FNSAVE приймач	Збереження повного стану FPU без очікування обробки виключних ситуацій
FXSAVE приймач	Швидке збереження повного стану FPU в приймачі. Дана команда не сумісна з командами FSAVE/FRSTOR
FRSTOR джерело	Відновлення повного стану FPU. Дана команда є зворотною до команди FSAVE
FXSTOR джерело	Швидке відновлення повного стану FPU. Дана команда є зворотною до команди FXSAVE
FLDENV джерело	Завантаження з джерела 5 допоміжних регістрів оточення співпроцесора (CWR, SWR, TWR, FIP, FDP)
FSTENV приймач	Збереження 5-ти допоміжних регістрів. Ця команда повністю еквівалентна команді WAIT FNSTENV і є оберненою до команди FLDENV
FNSTENV приймач	Збереження 5-ти допоміжних регістрів без очікування закінчення обробки виключних ситуацій. Дана команда є оберненою до команди FLDENV
FWAIT WAIT	Очікування готовності співпроцесора
FINIT	Ініціалізація співпроцесора. Дана команда повністю еквівалентна команді WAIT FNINIT
FNINIT	Ініціалізація співпроцесора без очікування обробки виключних ситуацій

Продовження табл.8.13

Команди управління співпроцесором

FENI (тільки в i8087)	Включення виключень. В старших версіях співпроцесора сприймається як команда FNOP.
FDISI (тільки в i8087)	Заборона виключень. В старших версіях співпроцесора сприймається як команда FNOP.
FNOP	Відсутність операції
FCLEX	Обнулення прапорців виключень в регістрі стану SWR (PE, UE, OE, ZE, DE, IE, ES, SE, B). Дана команда повністю еквівалентна команді WAIT FNCLEX
FNCLEX	Обнулення прапорців виключень без очікування
FINCSTP	Поле TOP регістра стану співпроцесора збільшується на 1
FDECSTP	Поле TOP регістра стану співпроцесора зменшується на 1
FFREE ST(i)	Звільнення регістра даних ST(i), i=0,…,7. В регістрі тегів TWR даних регістр позначається як пустий.

Основні особливості програмування

До основних особливостей програмування співпроцесора, пов'язаних з операндами, належать:

1. При анонімних зверненнях до пам'яті бажано задавати тип операнда через директиву **PTR**. Наприклад: FLD QWORD PTR[BX].

2. Консанти в операндах не допускаються. Їх портрібно задавати.

3. Наявність або відсутність операнда пов'язана не тільки з самою командою, але і з допустимими форматами команд співпроцесора. Наприклад, для арифметичних команд можливі декілька форматів.

Таблиця 8.14

Формати команд базової арифметики

Формат	**Мнемокод**	**Допустимі операнди**	**Приклад**
Стековий	F_{op}	ST(1), ST(0)	FSUB
Регістровий	F_{op}	ST(i), ST(0) ST(0), ST(i), i=0,…,7	FMUL ST(5), ST FADD ST,ST(2)
Регістровий з витягуванням із стека	$F_{op}P$	ST(i), ST(0), i=0,…,7	FMULP ST(3), ST(0)

Продовження табл. 8.14

Формати команд базової арифметики

Формат	Мнемокод	Допустимі операнди	Приклад
Дійсні дані в пам'яті	F_{op}	ST(0), mem_{32} ST(0), mem_{64}	FADD a
Цілі дані в пам'яті	FI_{op}	ST(0), mem_{16} ST(0), mem_{32} ST(0), mem_{64}	FIDIV k

Приклад: Написати програму для обчислення заданого змішаного арифметичного виразу для даних у форматі float, використовуючи команди співпроцесора. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Вираз:

$$x = \frac{\frac{c}{b} + a - \sqrt{24 + d}}{2ac - 1}$$

Лістинг програми:

```
#include "stdafx.h"
#include <stdio.h>
#include <math.h>
#include <conio.h>


int _tmain(int argc, _TCHAR* argv[])
{
	float a, b, c, d, res_c, res_asm, z = 24, x = 2, k = 1, n =
1;
	do
	{
		printf("Enter the values:\n");
		printf("a = "); scanf_s("%d", &a);
		printf("b = "); scanf_s("%d", &b);
		printf("c = "); scanf_s("%d", &c);
		printf("d = "); scanf_s("%d", &d);
	} while (a > 2147483647.0 || a < -2147483648.0 || b >
2147483647.0 || b < -2147483648.0 || c > 2147483647.0 || c < -
2147483648.0 || d > 2147483647.0 || d < -2147483648.0);

	if (b == 0)
	{
		printf("Error! Division by 0.\n");
```

```
        }

        else
        {
                res_c = (c / b + a - sqrt(24 + d)) / (2 * a * c - 1);
                printf("Result_C = %f\n", res_c);
        }
        __asm
        {
                finit  // ініціалізація співпроцесора
                fld c  // завантаження в вершину стека
                fdiv b // ділення (c/b)
                fld d  // завантаження в вершину стека
                fadd z // додавання (24+d)
                fsqrt  // квадратний корінь sqrt(24+d)
                fsub   // віднімання c/b-sqrt(24+d)
                fadd a // додавання c/b-sqrt(24+d)+a
                fld a  // завантаження в вершину стека
                fmul x // множення (2*a)
                fmul c // множення (2*a*c)
                fsub k // віднімання (2*a*c-1)
                fdiv   // ділення (c/b+a-sqrt(24+d))/(2*a*c-1)
                fstp res_asm // res_asm=(c/b+a-sqrt(24+d))/(2*a*c-1)
        }
        printf("Result_ASM = %f\n", res_asm);
        _getch();
        return 0;
}
```

Проведемо тестову перевірку роботи програми(рис.8.1).

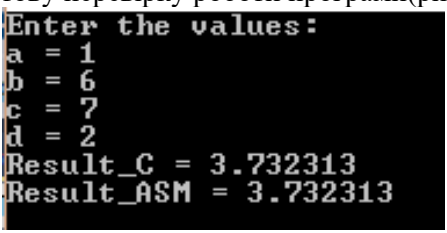

Рисунок 8.1 – Перевірка роботи програми

В даному виразі може виникнути помилка ділення на 0(рис.8.2):

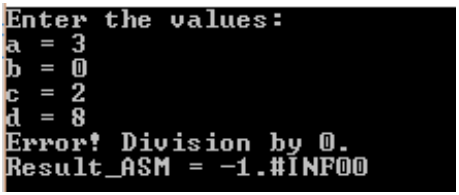

Рисунок 8.2 – Помилка при діленні на 0

Значення регістрів співпроцесора при пороковому виконанні

Команда	Опис	Стан регістрів	Коментар
finit	Ініціалізація співпроце-сора	cwr=037Fh; swr=0; twr=FFFFh; dpr=0; ipr=0;	
fld c	Завантаження в вершину стека	src	c
fdiv b	Ділення	dst=dst/src	c/b
fld d	Завантаження в вершину стека	src	d
fadd z	Додавання	dst=dst+src	d+24
fsqrt	Квадратний корінь	St(0)=	sqrt(24+d)
fsub	Віднімання	dst=dst-src	c/b-sqrt(24+d)
fadd a	Додавання	dst=dst+src	c/b-sqrt(24+d)+a
fld a	Завантаження в вершину стека	src	a
fmul x	Множення	dst=dst*src	2*a
fmul c	Множення	dst=dst*src	2*a*c
fsub k	Віднімання	dst=dst-src	2*a*c-1
fdiv	Ділення	dst=dst/src	(c/b-sqrt(24+d)+a)/ (2*a*c-1)
fstp res_asm	Збереження у вершині стеку	dst= St(0)	

Завдання на лабораторну роботу

1. Написати програму для ообчислення заданих змішаних арифметичних виразів(табл.8.15) для даних у форматі float, використовуючи команди співпроцесора. Виконати покрокове виконання асемблерного коду та навести стан регістрів при їх виконанні. Відмітити нормальні та аномальні результати, зробити аналіз результатів.

Таблиця 8.15

Дані для розрахунку

№ Варіанту	Вираз 1	Вираз 2
1	$\frac{2*c-d+\sqrt{23*a}}{\frac{a}{4}-1}$	$\frac{2*b-38*c}{\frac{arctg(b+a)}{c}+1}$
2	$\frac{c+4*d-\sqrt{123*c}}{1-\frac{a}{2}}$	$\frac{a+tg\left(\frac{b}{4}-1\right)}{\frac{c}{3}-a*d}$
3	$\frac{-2*c+d*82}{tg\left(\frac{q}{4}-1\right)}$	$\frac{\lg\left(25+2*\frac{a}{d}\right)}{c+a-b-1}$
4	$\frac{\lg(2*c-a)+d-152}{\frac{a}{4}+c}$	$\frac{\frac{b}{2}-\frac{53}{c}}{arctg(b-a)*c+1}$
5	$\frac{tg\left(a+\frac{c}{4}\right)-12*d}{a*b-1}$	$\frac{4*\ln\left(\frac{a}{b}\right)+1}{c+b-d+a}$
6	$\frac{-2*c-\sin\left(\frac{a}{d}\right)+53}{\frac{a}{4}-b}$	$\frac{\frac{4}{c}+tg(3*a)}{\frac{c}{a}-b-1}$
7	$\frac{2*c-\lg\left(\frac{d}{4}\right)}{a*a-1}$	$\frac{c+23-b+4}{a-\ln\left(a+\frac{c}{b}-1\right)}$
8	$\frac{tg(c)-d*23}{2*a-1}$	$\frac{arctg\left(\frac{c}{4}+28\right)*d}{\frac{a}{d}-c-1}$

9	$\frac{2*c-\frac{d}{23}}{\ln\left(1-\frac{a}{4}\right)}$	$\frac{\sqrt{2*b}-a+\frac{b}{c}}{a-\frac{c}{4}+1}$
10	$\frac{4*c+d-1}{c-tg\frac{a}{2}}$	$\frac{8*\lg(b-1)-c}{a*2+\frac{b}{c}}$
11	$\frac{2*c-d*\sqrt{\frac{42}{d}}}{c+a-1}$	$\frac{\frac{arctg(b-c)}{b}+\frac{a}{4}}{a+b-1}$
12	$\frac{\sqrt{\frac{25}{c}-d+2}}{d+a-1}$	$\frac{c*b-24+a}{b-\lg(2*c-1)+a}$
13	$\frac{arctg\left(c-\frac{d}{2}\right)}{2*a-1}$	$\frac{\frac{4*\ln(b)}{c}+1}{2*c+a-b}$
14	$\frac{4*\lg(c)-\frac{d}{2}+23}{a^2-1}$	$\frac{\sqrt{\frac{41*d}{a}+1}}{a-\frac{c}{b}+d}$
15	$\frac{c*tg(b+23)}{\frac{a}{2}-4*d-1}$	$\frac{\ln(a*b+2)*c}{41-\frac{b}{c}+1}$
16	$\frac{\frac{c}{d}+\ln\left(3*\frac{a}{2}\right)}{c-a+1}$	$\frac{arctg(a-c)*b+28}{4*\frac{b}{a}+1}$
17	$\frac{2*c+\lg(d)*51}{d-a-1}$	$\frac{b*a+\frac{c}{2}}{c-tg(b+1)}$
18	$\frac{2*c+\ln\left(\frac{d}{4}\right)+23}{a^2-1}$	$\frac{2*c+tg(a-21)}{\frac{c}{a}+d+1}$
19	$\frac{42*c-\frac{d}{2}+1}{a^2-\ln(b-5)}$	$\frac{\lg(4*b-c)*a}{b+\frac{c}{d}-1}$

20	$\frac{\frac{arctg(2*c)}{d}+2}{d-a-1}$	$\frac{1+a-\frac{b}{2}}{\sqrt{24+d-c}+\frac{a}{b}}$
21	$\frac{arctg\left(\frac{12}{c}\right)+73}{a^2-1}$	$\frac{a*\frac{b}{4}-1}{\sqrt{41-d}-b*a+c}$
22	$\frac{2*\frac{c}{a}-d^2}{d+tg(a-1)}$	$\frac{c-\frac{\ln(33+b)}{4}}{a-\frac{c}{b}-1}$
23	$\frac{\sqrt{\frac{53}{a}}+d-4*a}{1+a*c}$	$\frac{\lg(21-a)*\frac{c}{4}}{1+\frac{c}{a}+b}$
24	$\frac{\sqrt{15*a}+b-\frac{a}{4}}{c*d-1}$	$\frac{2*b-\ln(a+b)*c}{\frac{c}{4}-1}$
25	$\frac{-\frac{25}{a}+c-tg(b)}{1+c*\frac{d}{2}}$	$\frac{\lg\left(\frac{b}{a}+4\right)+d}{c-b+1}$
26	$\frac{\lg(4*a-1)+\frac{b}{2}}{d*c-5}$	$\frac{a-b*4-1}{\frac{c}{31}+tg(a*b)}$
27	$\frac{8*\lg(b+1)-c}{\frac{a}{2}+d*c}$	$\frac{c*4+28*\frac{d}{c}}{5-arctg(a*d-c-1)}$
28	$\frac{4*a-\ln(b-1)}{\frac{c}{d}+18}$	$\frac{41-\frac{d}{4}-1}{\frac{c}{tg(b+a)}-d}$
29	$\frac{\frac{arctg(4*b)}{c}-1}{12+a-b}$	$\frac{\frac{c}{b}-\sqrt{24+d}+a}{2*a*c-1}$

30	$\frac{arctg(b)+c*b-\frac{a}{4}}{a*d-1}$	$\frac{a+\frac{c}{b}-\sqrt{28*d}}{4*b*a+1}$

Контрольні питання

1. Що таке математичний співпроцесор?
2. Типи звичайних даних співпроцесора.
3. Спеціальних формати даних співпроцесора.
4. Назвіть регістри співпроцесора.
5. Звичайні і реверсні операції.
6. Команди пересилки даних.
7. Команди завантаження констант.
8. Арифметичні команди
9. Трансценденті операції
10. Команди порівняння та управління співпроцесором

СПИСОК ЛІТЕРАТУРИ

1. Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения. 2-е изд. испр. и доп. / Н.Г. Голубь. – СПб.: ООО "ДиаСофтЮП", 2002. – 656 с.URL (веб-посилання)

2. Юров В.И. Assembler. Учебник для вузов. 2-е изд. / В.И. Юров. – СПб.: Питер, 2010. – 637 с.URL (веб-посилання)

3. Юров В.И. Assembler. Учебник для вузов. / В.И. Юров. – СПб.: Питер, 2003. – 637 с.URL (веб-посилання)

4. Юров В.И. Assembler: Практикум. / В.И. Юров. – СПб.: Питер, 2003. – 400 с.URL (веб-посилання)

5. Пирогов В.Ю. Assembler. Учебный курс. / В.Ю. Пирогов. – М.: Издатель Молгачева С.В., Издательство Нолидж, 2001. – 848 с.URL (веб-посилання)

6. Магда Ю.С. Ассемблер для процессоров Intel Pentium. / Ю.С. Магда. – СПб.: Питер, 2006. – 400 с.URL (веб-посилання)

7. Рудольф Марек. Ассемблер на примерах. Базовый курс. / Марек Рудольф. – СПб: Наука и Техника, 2005. – 240 с.URL (веб-посилання)

8. Ирвин, Кип. Язык ассемблера для процессоров Intel, 4-е издание.: Пер. с англ. / Кип Ирвин. – М.: Издательськи дом «Вильямс», 2005. – 912 с.URL (веб-посилання)

9. Абель П. Язык Ассемблера для IBM PC и программирования / Пер. с англ. Ю.В. Сальникова. - М.: Высш. шк., 1992. 447 с.

Для нотаток

Для нотаток

Н а в ч а л ь н о - м е т о д и ч н е в и д а н н я

АРХІТЕКТУРА КОМП'ЮТЕРА

Частина 2

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
ДЛЯ ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Підготували
Єфіменко Андрій Анатолійович, Байлюк Єлизавета Максимівна, Покотило Олександра Андріївна

Редактори
Комп'ютерне верстання –

Свідоцтво про реєстрацію № __ від _____ 201_ року
Підписано до друку __.__.16. Формат 60×84/16.
Ум. друк. арк. ____. Зам. __ офс.

Безкоштовно

Друкарня ЖДТУ