Лабораторна робота № 4
ОБЧИСЛЕННЯ ПРОСТИХ ТА СКЛАДНИХ 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 та складних 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
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 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.

Основні регістри процесорів Intel наведені в табл. 4.3.

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
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

Група регістрів EAX, EBX, ECX, EDX – це регістри загального призначення, або регістри даних. Користувач може використовувати їх на свій розсуд для тимчасового зберігання будь-яких об'єктів (даних або адрес) і виконання над ними необхідних операцій. При цьому регістри допускають незалежне звернення до своїх молодших частин.

Зважаючи на те, що раніше 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 наведено список регістрів загального призначення, які поділені на «половинки» і «четвертинки» і до цих «маленьких» регістрів можна звертатися в програмі як до окремих регістрів.

Таблиця 4.4

Список регістрів загального призначення

<table>
<tr><th rowspan="2">Регістр</th><th>Старші розряди</th><th colspan="2">Імена 16 та 8 бітних регістрів</th></tr>
<tr><th>31…16</th><th>15…8</th><th>7…0</th></tr>
<tr><td rowspan="2">EAX</td><td rowspan="2">…</td><td colspan="2">AX</td></tr>
<tr><td>AH</td><td>AL</td></tr>
<tr><td rowspan="2">EBX</td><td rowspan="2">…</td><td colspan="2">BX</td></tr>
<tr><td>BH</td><td>BL</td></tr>
<tr><td rowspan="2">ECX</td><td rowspan="2">…</td><td colspan="2">CX</td></tr>
<tr><td>CH</td><td>CL</td></tr>
<tr><td rowspan="2">EDX</td><td rowspan="2">…</td><td colspan="2">DX</td></tr>
<tr><td>DH</td><td>DL</td></tr>
<tr><td>ESI</td><td>…</td><td colspan="2">SI</td></tr>
<tr><td>EDI</td><td>…</td><td colspan="2">DI</td></tr>
<tr><td>EBP</td><td>…</td><td colspan="2">BP</td></tr>
<tr><td>ESP</td><td>…</td><td colspan="2">SP</td></tr>
<tr><td>EIP</td><td>…</td><td colspan="2">IP</td></tr>
</table>

Приклад розділення регістра EAX на молодші частини показано на рис.4.1.

<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>

Рис. 4.1 – Приклад регістра

Регістри EBP, ESP, ESI, EDI – це також регістри загального призначення але на відміну від регістрів даних, не допускають побайтову адресацію. В них також можна зберігати призначені для користувача дані, але робити це потрібно більш обережно, щоб не отримати «несподіваний» результат.

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.

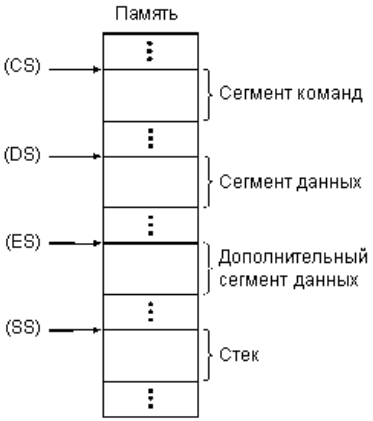

Рис. 4.2 – Сегменти команд, даних і стека в пам'яті

Окремого розглянемо регістр прапорів і сегментні регістри.

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.

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

Приймач: = <Джерело>

ДОВЖИНИ ОБОХ операндів повинні бути однаковими. Якщо ж вони різні, 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 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 викристовуються 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 *прапорець перенесення*	Результат додавання не помістився в операнд-приймач
PF=1 *прапорець парності*	Результат додавання має парну кількість бітів із значенням 1
SF=1 *праорець знаку*	Копіюється СТАРШИЙ (ЗНАКОВИЙ) біт результату додавання
ZF=1 *прапорець нуля*	Результат додавання дорівнює нулю
OF=1 *прапорець переповнення*	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 ADC

Команда **ADC** (ADdition with Carry) – додавання з перенесенням, відрізняється від команди ADD використанням біта переносу **CF** при додаванні. Тому вона може використовуватися при складанні багатобайтних цілих чисел, довжина яких більша, ніж розрядність регістрів процесора який використовується.

Синтаксис:

ADC Приймач, Джерело

Логіка роботи:

<Приймач> = <Приймач> + <Джерело> + <CF>

На рис 4.3 показано додавання двох двійкових чисел командою ADD:

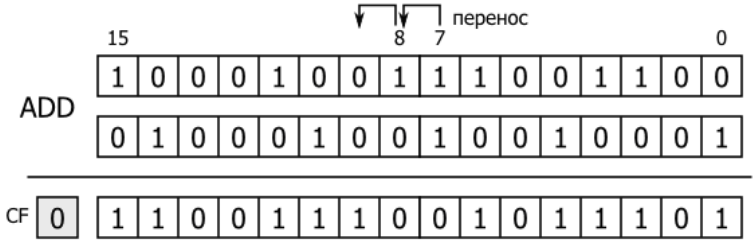

Рис. 4.3 – Додавання двох 16-розрядних чисел 35276 + 17553 = 52829 командою ADD.

При додаванні відбувається перенесення з 7-го розряду в 8-й, саме на межі між байтами. Якщо ми будемо складати ці числа частинами командою ADD, то перенесений біт загубиться і в результаті ми отримаємо помилку. На щастя, перенесення зі старшого розряду завжди зберігається у прапорі CF. Щоб додати цей перенесений біт, достатньо застосувати команду ADC. Зазвичай ця команда працює у парі з командою ADD (складання молодших частин числа).

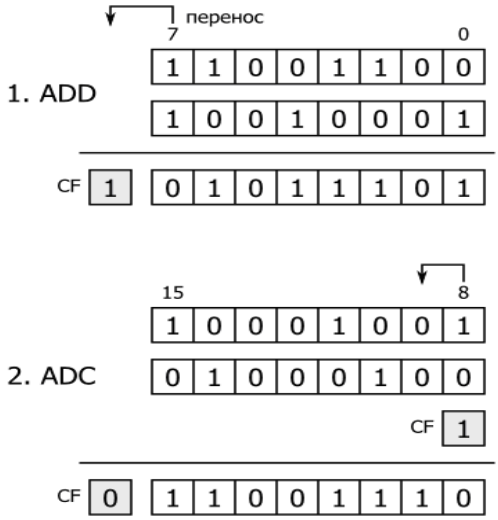

Рис. 4.4 – Додавання двох 16-розрядних чисел 35276 + 17553 = 52829 командою ADC.

1.3 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.
```

2.2 Команда SBB

Команда SBB (SuBtract with Borrow) CF – віднімання із перенесенням (позикою) прапора CF. Прцює аналогічно команді ADC.

Синтаксис:

SBB Приймач, Джерело

Логіка роботи:

<Приймач> = <Приймач> – <Джерело> – <CF>

2.3 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
```

2.3 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 <iostream>
#include <stdio.h>
#include <cstdlib>

int main()
{
	signed char a, b, c, d, res_c, res_asm;
	printf("a = "); scanf_s("%hhd", &a);
	printf("b = "); scanf_s("%hhd", &b);
	printf("c = "); scanf_s("%hhd", &c);
	printf("d = "); scanf_s("%hhd", &d);
	res_c = -(b + c + d) - (a + 1);
	printf("Result C = %hhd\n", res_c);
	__asm {
		mov al, b;    // <al> = b
		mov cl, c;    // <cl> = c
		add al, cl;   // <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= %hhd\n", res_asm);
	system("Pause");
	return 0;
}
```

Проведемо тестові перевірки роботи програми(рис.4.5 – рис.4.6)

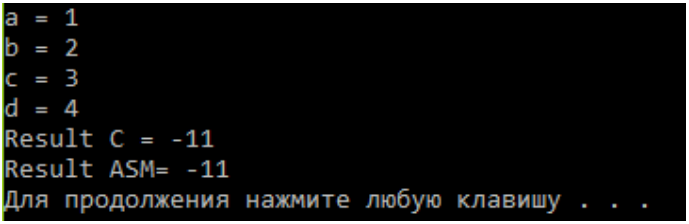

Рис. 4.5 – Перевірка роботи програми для 8-бітних вхідних даних та 8-бітного результату

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	mov cl, c	н/в	н/в	3	н/в	–
3	add al, cl	5	н/в	н/в	н/в	–
4	add al, d	9	н/в	н/в	н/в	–
5	neg al	-9	н/в	н/в	н/в	–
6	mov cl, a	н/в	н/в	1	н/в	–
7	inc cl	н/в	н/в	2	н/в	–
8	sub al, cl	-11	н/в	н/в	н/в	–
9	mov res_asm, al	-11	н/в	н/в	н/в	–

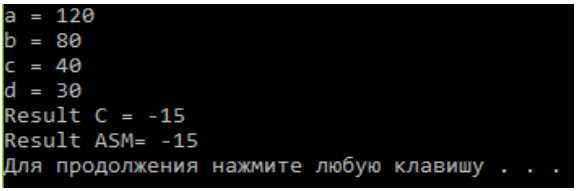

Рис. 4.6 – Перевірка роботи програми для 8-бітних значень вхідних даних та результату, що перевищує 8 біт

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	mov cl, c	н/в	н/в	40	н/в	–
3	add al, cl	120	н/в	н/в	н/в	–
4	add al, d	-106	н/в	н/в	н/в	CF=1, OF=1
5	neg al	106	н/в	н/в	н/в	CF=1, OF=1
6	mov cl, a	н/в	н/в	120	н/в	–
7	inc cl	н/в	н/в	121	н/в	–
8	sub al, cl	-15	н/в	н/в	н/в	–
9	mov res_asm, al	-15	н/в	н/в	н/в	–

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

3 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 4.7

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.

4 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 4.8

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.

5 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 4.9

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)		

Таблиця 4.10

Логіка роботи команди CWD при отриманні знакового діленого

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>		

Ці команди також застосовуються при вирівнюванні операндів по довжині.

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, INC, SUB, DEC, NEG, IMUL, IDIV, CBW, CWD, CDQ.

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 <iostream>
#include <stdio.h>
#include <cstdlib>

int main()
{
	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("%hhd", &a);
	printf("c = "); scanf_s("%hhd", &c);
	printf("d = "); scanf_s("%hhd", &d);
	res_c = (c / 4 - d * 62) / (a * a + 1);
	printf("Result C = %hhd\n", res_c);
```

```
    __asm {
          mov al, c;        // <al> = c
          cbw;              // <ax> = <al> = c
          mov bl, 4;        // <bl> = 4
          idiv bl;          // <al> = <ax>/<bl> = c/4
          mov bl, al;       // <bl> = c/4
          mov al, d;        // <al> = d
          mov cl, 62;       // <cl> = 62
          imul cl;          // <ax> = <al>*<cl> = d*62
          mov cl, 1;        // <cl> = 1
          idiv cl;          // <al> = <ax>/<cl> = d*62/1
          mov dl, al;       // <dl> = d*62
          sub bl, dl;       // <bl> = c/4-d*62
          mov al, a;        // <al> = a
          imul al;          // <ax> = <al>*<al> = a*a
          idiv cl;          // <al> = a*a
          inc al;           // <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> = <ax>/<cl>=(c/4-d*62)/(a*a+1)
          mov res_asm, al; // res_asm =  <al>
    }
    printf("Result ASM = %hhd\n", res_asm);
    system("Pause");
    return 0;

}
```

Проведемо тестову перевірку роботи програми(рис. 4.7)

```
(c/4-d*62)/(a*a+1)
Enter the values of the range [-128...127]
a = 1
c = 8
d = 1
Result C = -30
Result ASM = -30
Для продолжения нажмите любую клавишу . . .
```

Рис. 4.7 – Перевірка роботи програми для 8-бітних даних та 8-бітного результату

Значення регістрів при покроковому виконанні

Крок	Команда	Значення регістра					EFLAGS/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	–
17	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. 4.8).

```
(c/4-d*62)/(a*a+1)
Enter the values of the range [-128...127]:
a = 130
c = 4
d = 1
```

Рис. 4.8 – Перевірка роботи програми для даних, що виходять за межі діапазону допустимих значень

В результаті виникає помилка (рис. 4.9).

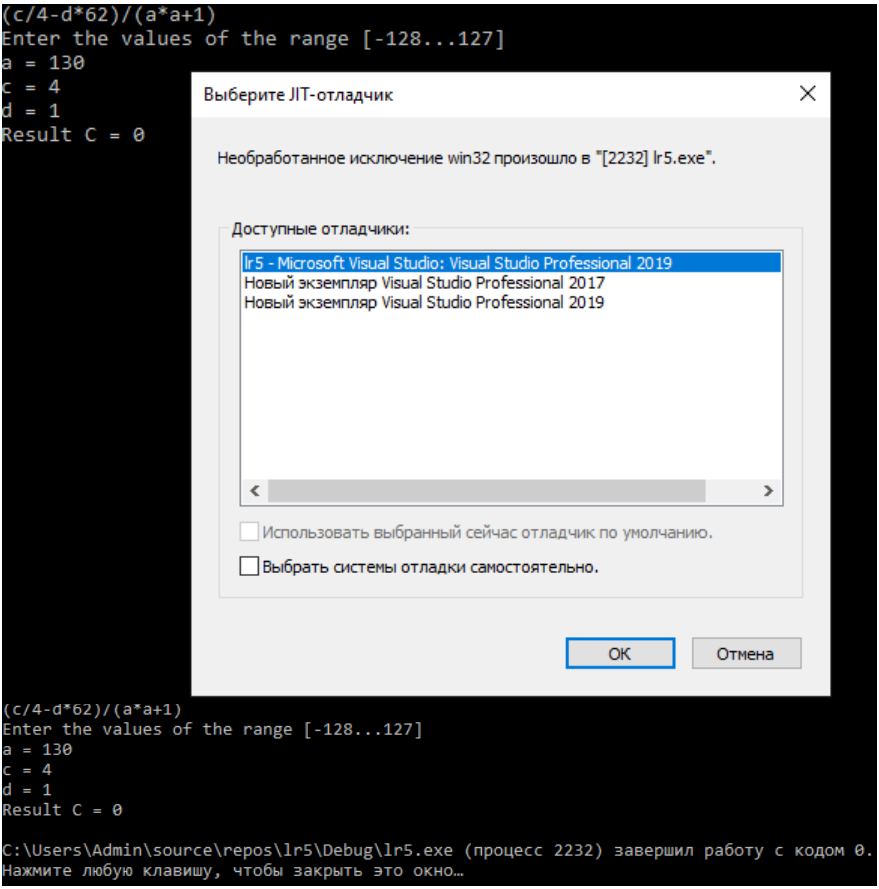

Рис. 4.9 – Помилка при введенні даних, що виходять за межі діапазону допустимих значень

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.4.11) для початкових даних в знаковому форматі довжиною за варіантом, використовуючи прості арифметичні операції ADD, INC, SUB, DEC, NEG. Виконати покрокове виконання асемблерного коду та навести значення регістрів при їх виконанні для нормальних та значень які перевищують межі діапазону допустимих.

2. Написати програму для обчислення заданого цілочисленого виразу (табл. 4.12) для початкових даних в знаковому форматі довжиною за варіантом, використовуючи складні 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 4.11

Дані для розрахунку

№ Вар-ту	Вираз	Формат представлення
1	–(a+b–c)–(d+1)–(e–f)	16 біт
2	1–(b–d+c)+(a+f)–e+2	32 біти
3	–(b+c+d–f)–(a+1–e)–1	16 біт
4	f–(a+c–b+e)–(d+2)+3	32 біти
5	–(b–1+c–1)+(a–e+d+f)	16 біт
6	(b–1)+(a+1)–c–d–(f–e)	32 біти
7	12–(b–c–a)+d–(e+1)+f	16 біт
8	(a+b–e)–1–(c+d)–(f+3)	32 біти
9	–(a+b+c–1–d)–(f+e+1)	16 біт
10	–(a+b–c–d)+2+e–f+2	32 біти
11	b+c+d–(a+4)–(e+1)+f	16 біт
12	c+d–(a+b+1)–(f+1–e)	32 біти
13	(d–c)+(b–a)–1+(e–1+f)	16 біт
14	(d+c–1)+(a+b–1)–(f–e)	32 біти
15	–(a+b–1)+(c–d)+e–f+6	16 біт
16	(a+b)+3–(d–c–1)–(e+f)	32 біти
17	(b–1)+(a+1)–c+d–(f+e)	16 біт
18	–(a+c–b)–(d–1)+4–f+e	32 біти
19	(a+1)+b+c–d–2–(f+1–e)	16 біт
20	b+c+d–a+2–(e+1)–(f–1)	32 біти
21	–(a+1)–b+c+d–f–(e+1)	16 біт
22	–(b+2)–c+a–d+e–(f+1)	32 біти
23	–(c–2)+a+b+d–(e+1)–f	16 біт
24	–(d+2)+a–e–b+c+(f–1)	32 біти
25	–(a+b)–(d–c+1)+(f–e)–1	16 біт
26	a+1–(c+d–b)+(f+1–e)	32 біти
27	c–1–(a–b+d)–(e–1)+f	16 біт
28	c–3+a+b–(d+1)+e–(f+1)	32 біти
29	–(2+a–b)+c+d–(e+1)+f	16 біт
30	–(b–a)+c+d+2–(f–1+e)	32 біти

Таблиця 4.12

Дані для розрахунку

№ Вар-ту	Вираз	Формат представлення
1	(c+4*d-123+e)/(1-a/2+f)	32 біти
2	(2*c+d-52+e)/(a/4+1-f)	16 біт
3	(-2*c-d+53-e)/(a/4-1+f)	32 біти
4	(2+c-d*23-e)/(2*a*a-1-f)	16 біт
5	(4*c+d-1+e)/(c-a/2+f)	32 біти
6	(25/c-d+2+e)/(b+a*a-1-f)	16 біт
7	(4*c-d/2+23-e)/(b+a*a-1+f)	32 біти
8	(c/d+3*a/2-e)/(c-a+1-f)	16 біт
9	(2*c+4/d+23+e)/(a*a-1+f)	32 біти
10	(2*c/d+2+e)/(d-a*a-1-f)	16 біт
11	(2*c/a-d*d-e)/(d+a-1+f)	32 біти
12	(-15*a+b-a/4-e)/(b*a-1-f)	16 біт
13	(4*a-1+b/2+e)/(b*c-5+f)	32 біти
14	(4*a-b-1+e)/(c/b+a-f)	16 біт
15	(b+c*b-a/4-e)/(a*b-1+f)	32 біти
16	(c/b-24+a-e)/(2*a*c-1-f)	16 біт
17	(41-d/4-1+e)/(c/b+a*d+f)	32 біти
18	(b/a+4*c+e)/(c-b+1-f)	16 біт
19	(c-33+b/4-e)/(a*c/b-1+f)	32 біти
20	(c/4+28*d-e)/(a/d-c-1-e)	16 біт
21	(1+6*a-b/2+e)/(c+a/d+f)	32 біти
22	(4*b-c*a+e)/(b+c/28-1-f)	16 біт
23	(4/c+3*a-e)/(c/a-b-1+f)	32 біти
24	(4*a/b+1-e)/(c*b-18+a-f)	16 біт
25	(b-c/b+a/4+e)/(a*b-1+f)	32 біти
26	(c*b-24+a+e)/(b/2*c-1-f)	16 біт
27	(41*d/4+1-e)/(a-c/b+a*d+f)	32 біти
28	(b*a+c/2-e)/(4*c-b+1-f)	16 біт
29	(c+23-b*4+e)/(a*c/b-1+f)	32 біти
30	(c*4+28/d-e)/(a*d-c-1+f)	16 біт

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.
11. Назвіть складні команди цілочисельної арифметики мови Assembler.
12. Команда множення MUL та IMUL. Призначення та синтаксис.
13. Команда ділення DIV та IDIV. Призначення та синтаксис.
14. Команда розширення знаку CBW та CWD. Призначення та синтаксис.
15. Неявні операнди команд MUL, IMUL.
16. Неявні операнди команд DIV, IDIV.
17. Логіка роботи команди CBW при отриманні знакового діленого.