СПІЛЬНЕ ВИКОРИСТАННЯ МОВИ ASSEMBLER ТА МОВ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ

2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аероскосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

СПІЛЬНЕ ВИКОРИСТАННЯ МОВИ ASSEMBLER ТА МОВ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ

Навчальний посібник

Харків «ХАІ» 2021

УДК 004.413
М51

Колектив авторів:
Є. С. Меняйлов, К. О. Базілевич, І. О. Трофимова,
Д. І. Чумаченко, П. А. Пирогов

Рецензенти: д-р техн. наук, проф. В. О. Тімофєєв,
канд. техн. наук, доц. І. Т. Зарецька

Меняйлов, Є. С.

М51 Спільне використання мови Assembler та мов програмування високого рівня [Електронний ресурс] : навч. посіб. /Є. С. Меняйлов, К. О. Базилевич, І. О. Трофимова та ін. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2021. – 116 с.

Описано лабораторні роботи для вивчення й закріплення матеріалу з дисципліни «Архітектура обчислювальних систем», наведено пояснювальний матеріал, приклади розв'язання типових задач і рекомендації щодо їх виконання.

Для студентів усіх спеціальностей технічних вузів.

Іл. 60. Табл. 19. Бібліогр.: 6 назв

УДК 004.413

ВСТУП

Assembler – мова програмування низького рівня для програмованої обчислювальної системи (мікропроцесора, мікроконтролера, комп'ютера або іншого програмованого пристрою), в якій існує сувора відповідність між операторами мови та машинними командами [2].

Команди мови Assembler відповідають машинним кодам відповідного мікропроцесора чи мікроконтролера. Фактично, мова Assembler являє собою зручнішу символьну форму запису машинних команд. Як наслідок, програми, написані для одного типу процесорів, на іншому не будуть функціонувати. Мова Assembler також містить засоби для створення міток і переходів, що необхідно для створення циклів і розгалужень. Можуть бути наявні засоби для створення макросів, процедур. Кожне сімейство (модельний ряд) мікропроцесорів має свій набір команд і, відповідно, свій набір інструкцій мовою Assembler.

Мови Assembler так само, як і використання слова assembly, беруть свій початок від перших ЕОМ зі збереженням програми. Одна з перших мов Assembler була розроблена у 1947 році Катлін Бут (Kathleen Booth) для машини ARC2 в університеті Біркбек (Лондон), після консультацій з Джоном фон Нойманом і Германом Ґолдстіном у Інституті перспективних досліджень [5, 6]. Машина EDSAC у 1949 році мала Assembler під назвою initial orders з однолітерними мнемоніками [7]. Для машин IBM 650 існувала програма SOAP (Symbolic Optimal Assembly Program), написана Стеном Полі (Stan Poley) у 1955 р. [8].

Мови Assembler дозволили вивільнити програмістів від надзвичайно низькорівневих, рутинних і утомливих «ручних» процедур, яких потребували перші комп'ютери – запам'ятовування людиною кодів команд, адрес, ручного обчислення адрес переходів (і зміни їх при кожному додаванні чи вилученні команд), подання чисел тощо. Приблизно до середини 1980-х років мови Assembler широко використовувалися у програмуванні – як для великих ЕОМ (мейнфреймів), так і для персональних і мікрокомп'ютерів. Утім, прогрес у розвитку процесорів і пам'яті спричинив активний прогрес методів компіляції з високорівневих мов і появу компіляторів, що значно підвищували продуктивність роботи програміста. У XXI столітті мови Assembler використовуються там, де потрібні пряме керування апаратурою, доступ до спеціалізованих інструкцій процесора чи співпроцесорів, або для кодування критичних (до часу чи розміру) секцій програми. Типовими сферами застосування Assembler є драйвери пристроїв, убудовані системи, системи реального часу, ядра ОС, резидентні монітори тощо.

Багато важливих програм було написано цілком мовою Assembler (лише у 1961 р. з'явилася перша операційна система Burroughs MCP, написана частково мовою високого рівня (алголоподібний ESPOL). Велика кількість комерційного програмного забезпечення для мейнфреймів IBM також написана мовою Assembler.

Більшість ранніх мікрокомп'ютерів, обмежених оперативною пам'яттю, потребувало програмного забезпечення, написаного вручну майже виключно мовою Assembler, включно з кодом BIOS, дискової операційної системи і великої кількості прикладних програм. Типовими прикладами великих Assemblerних програм 1970 – 80-х років є BIOS і DOS для комп'ютерів IBM PC, компілятор Turbo Pascal, перші програми електронних таблиць (наприклад, Lotus 1-2-3), численні комп'ютерні ігри.

З часу винаходу перших компіляторів відбувалися дебати щодо доцільності використання мови Assembler у програмуванні прикладних задач. Утім, системне програмування завжди лишалося особливою нішею, де Assembler був і залишається важливим. Фундаментальні поняття, такі як двійкова арифметика, керування пам'яттю, стек, кодування символів, переривання, проектування компіляторів тощо можуть бути повністю осягнені людиною лише при ґрунтовному розумінні роботи комп'ютера на апаратному рівні, що неодмінно передбачає розуміння набору команд і адресації процесора.

Приклади застосувань:

– гіпервізори;

– реалізація криптографічних та інших специфічних алгоритмів (наприклад, цифрового перетворення сигналів);

– фрагменти ядер ОС або цілі мікроядра;

– початкові завантажувачі, Firmware, резидентні монітори, BIOS;

– результат роботи компілятора (коли компілятор генерує не об'єктний код, а текст мовою Assembler), який можна переглянути і змінити за потребою;

– inline-інтегрування команд мовою Assembler безпосередньо у тіло програми, написаної мовою високого чи середнього рівня (C, Паскаль);

– низькорівневі віруси чи руткіти;

– машинний код, що модифікує сам себе;

– реверс-інжиніринг;

– програми, повністю незалежні від основних системних бібліотек (таких, як libc чи crt0);

– різні типи оптимізації, що з тих чи інших причин недоступні при кодуванні мовою високого рівня.

Лабораторна робота № 1

ПЕРЕВЕДЕННЯ ЧИСЕЛ З ОДНОЇ СИСТЕМИ ЧИСЛЕННЯ В ІНШУ

Мета роботи: вивчити правила переведення чисел з одної системи числення у іншу; ознайомитися з особливостями використання систем числення з основою, що є степенем числа 2.

Завдання

1. Перевести цілі десяткові числа у іншу систему числення, і навпаки.
2. Виконати арифметичні операції над парами чисел у різних системах числення.
3. Перевести дробові десяткові числа у іншу систему числення.
4. Перевести цілі двійкові числа у систему числення з основою, що є степенем числа 2, і навпаки методом групування розрядів.
5. Запрограмувати необхідні завдання.
6. Скласти звіт, який повинен містити тему, мету роботи, завдання, перелік правил для виконання операції, результати виконання завдань без використання програми, програмний код, результати виконання програми.

Теоретичні відомості

Системою числення називається сукупність прийомів найменування і запису чисел. У будь-якій системі числення для подання чисел вибираються деякі символи (їх називають цифрами), а інші числа виходять в результаті будь-яких операцій над цифрами даної системи числення.

Система називається позиційною, якщо значення кожної цифри (її вага) змінюється залежно від її положення (позиції) у послідовності цифр, що зображують число.

Число одиниць будь-якого розряду, що об'єднуються в одиницю більше старшого розряду, називають основою позиційної системи числення. Якщо кількість таких цифр P, то система числення називається P-ричною. Основа системи числення збігається з кількістю цифр, використовуваних для запису чисел у цій системі числення.

Запис довільного числа x у P-ричній позиційній системі числення ґрунтується на поданні цього числа у вигляді полінома

$$x = a_n P^n + a_{n-1} P^{n-1} + \ldots + a_1 P^1 + a_0 P^0 + a_{-1} P^{-1} + \ldots + a_{-m} P^{-m}.$$

Арифметичні дії над числами у будь-який позиційній системі числення виконуються за тими ж правилами, що і в десятковій системі, оскільки всі вони ґрунтуються на правилах виконання дій над відповідними

поліномами. При цьому потрібно користуватися таблицями додавання і множення, що відповідають основі *P*-ричної системи числення.

При перекладі чисел з десяткової системи числення в систему з основою *P* >1 зазвичай використовують такий алгоритм:

1) якщо перекладається ціла частина числа, то вона ділиться на *P*, після чого запам'ятовується залишок від ділення. Отримана частка знову ділиться на *P*, залишок запам'ятовується. Процедура триває доки частка не стане такою, що дорівнює нулю. Залишки від ділення на *P* виписуються в порядку, зворотному їх отриманню;

2) якщо перекладається дробова частина числа, то вона множиться на *P*, після чого ціла частина запам'ятовується і відкидається. Знову отримана дробова частина множиться на *P,* і т. д. Процедура триває доки дробова частина не стане такою, що дорівнює нулю. Цілі частини виписуються після двійкової коми в порядку їх отримання. Результатом може бути або кінцевий, або періодичний двійковий дріб. Коли дріб є періодичним, доводиться обривати множення на будь-якому етапі і задовольнятися наближеним записом вихідного числа в системі з основою *P*.

Приклад 1

Перевести числа з десяткової системи числення в двійкову (отримати п'ять знаків після коми в двійковому поданні):

$$а – 464_{(10)}; б – 380{,}1875_{(10)}; в – 115{,}94_{(10)}.$$

Розв’язання

	464	| 0		380	| 0		|	1875		115	| 1		|	94
	232	| 0		190	| 0		0|	375		57	| 1		1|	88
	116	| 0		95	| 1		0|	75		28	| 0		1|	76
	58	| 0		47	| 1		1|	5		14	| 0		1|	52
а	29	| 1	б	23	| 1		1|	0	в	7	| 1		1|	04
	14	| 0		11	| 1					3	| 1		0|	08
	7	| 1		5	| 1					1	| 1		0|	16
	3	| 1		2	| 0									
	1	| 1		1	| 1									

Відповідь

$а – 464_{(10)} = 111010000_{(2)}$;

$б – 380{,}1875_{(10)} = 101111100{,}0011_{(2)}$;

$в – 115{,}94_{(10)} \approx 1110011{,}11110_{(2)}$ (у цьому випадку було отримано шість знаків після коми, після чого результат був округлений).

Якщо необхідно перевести число з двійкової системи числення у систему числення, основою якої є степінь двійки, досить об'єднати цифри

двійкового числа в групи по стільки цифр, який показник степеня, і використовувати наведений нижче алгоритм. Наприклад, якщо переклад здійснюється у вісімкову систему, то групи будуть містити три цифри ($8 = 2^3$). Отже, в цілій частині будемо виконувати групування справа наліво, в дробовій – зліва направо. Якщо в останній групі бракує цифр, дописуємо нулі: в цілій частині – зліва, у дробовій – справа. Потім кожна група замінюється відповідною цифрою нової системи. Відповідності наведені в таких таблицях:

P	2	00	01	10	11
	44	00	11	22	33

P	2	000	001	010	011	100	101	110	111
	8	0	1	2	3	4	5	6	7

P	2	0000	0001	0010	0011	0100	0101	0110	0111
	16	0	1	2	3	4	5	6	7
	2	1000	1001	1010	1011	1100	1101	1110	1111
	16	8	9	A	B	C	D	E	F

Приклад 2

Перевести з двійкової системи в шістнадцяткову число $1111010101{,}11_{(2)}$.

Розв’язання

$$0011\ 1101\ 0101{,}1100_{(2)} = 3D5{,}C_{(16)}.$$

При переведенні чисел із системи числення з основою *P* у десяткову систему числення необхідно пронумерувати розряди цілої частини справа наліво, починаючи з нульового, і в дробовій частині, починаючи з розряду відразу після коми зліва направо (початковий номер -1). Потім обчислити суму добутків відповідних значень розрядів на основу системи числення в степені, що дорівнює номеру розряду. Це і є подання вихідного числа в десятковій системі числення.

Приклад 3

Перевести число в десяткову систему числення:

а – $1000001_{(2)}$; б – $1000011111{,}0101_{(2)}$; в – $1216{,}04_{(8)}$; г – $29A{,}5_{(16}$.

Розв’язання

а – $1000001_{(2)} = 1\text{-}\ 2^6+0 \cdot 2^5+0 \cdot 2^4+0 \cdot 2^3+0 \cdot 2^2+ 0 \cdot 2^1+1 \cdot 2^0 = 64+1= 65_{(10)}$.

Зауваження. Очевидно, якщо в будь-якому розряді стоїть нуль, то відповідний доданок можна опускати;

б – $1000011111{,}0101_{(2)} = 1\cdot2^9 + 1\cdot2^4 + 1\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-2} + 1\cdot2^{-4} =$
$= 512 + 16 + 8 + 4 + 2 + 1 + 0{,}25 + 0{,}0625 = 543{,}3125_{(10)}$;

в – $1216{,}04_{(8)} = 1\cdot8^3+2\cdot8^2+1\cdot8^1+6\cdot8^0+4\cdot 8^{-2} = 512 + 128 + 8 + 6 + 0{,}0625 =$
$= 654{,}0625_{(10)}$;

г – $29A{,}5_{(16)} = 2\cdot16^2 + 9\cdot16^1 + 10\cdot16^0 +5\cdot16^{-1} = 512+144+10+0{,}3125 =$
$= 656{,}3125_{(10)}$.

Варіанти завдань

Запрограмувати виконання таких операцій і навести приклади виконання завдань без використання програмного забезпечення:

– перевести вісім трирозрядних цілих десяткових чисел у 3-, 4-, 5-ричні системи числення, і навпаки;

– виконати арифметичні операції «+», «*» над чотирма парами чисел у 3-, 4-, 5-ричних системах числення;

– перевести чотири дробових десяткових числа у 8- і 16-ричні системи числення;

– перевести вісім 16-розрядних цілих двійкових чисел у 8- і 16-ричні системи числення і у зворотньому напрямку методом групування розрядів;

– виконати арифметичні операції «+», «*» над чотирма парами чисел у 8- та 16-ричних системах числення.

Лабораторна робота № 2

КОДУВАННЯ ІНФОРМАЦІЇ В ОБЧИСЛЮВАЛЬНИХ МАШИНАХ

Мета роботи: вивчити подання чисел у форматі з фіксованою точкою (комою) для цілих чисел і у форматі з плаваючою точкою (комою) для дійсних чисел, а також побудову внутрішнього подання чисел, символів, зображень у пам’яті обчислювальних машин.

Завдання

1. Подати цілі числа у форматі з фіксованою точкою (комою) і дробові числа у форматі з плаваючою точкою (комою).

2. Побудувати форму внутрішнього подання цілого числа.

3. Перевести двійкові числа у строки десяткових цифрових символів.

4. Побудувати форму внутрішнього подання зображення у пам’яті.

5. Скласти звіт, який повинен містити тему, мету роботи, завдання, перелік правил для виконання операції, результати виконання завдань без використання програми, програмний код, результати виконання програми.

Теоретичні відомості

Усі числові дані зберігаються в машині в двійковому вигляді, тобто у вигляді послідовності нулів та одиниць, однак форми зберігання цілих і дійсних чисел різні.

Для подання чисел у пам'яті ПК використовуються два формати: формат з фіксованою точкою (комою) для цілих чисел і формат з плаваючою точкою (комою) для дійсних чисел.

Розглянемо подання цілих чисел. Множина цілих чисел, поданих в ЕОМ, обмежена. Діапазон значень залежить від розміру комірок пам'яті, використовуваних для їх зберігання.

Для цілих чисел існують два подання: беззнакове і зі знаком.

У К-розрядній комірці може зберігатися 2^K різних значень цілих чисел. Діапазон значень цілих беззнакових (тільки позитивних) чисел –

від 0 до 2^K - 1.

Для 16-розрядної комірки діапазон становить від 0 до 65535, для 8-розрядної комірки – від 0 до 255.

Діапазон значень цілих чисел зі знаком (і негативні, і позитивні в рівній кількості) – від 2^{K-1} до 2^{K-1}-1.

Для 16-розрядної комірки діапазон становить від -32768 до 32767, для 8-розрядної комірки – від -128 до 127.

Щоб отримати внутрішнє подання цілого позитивного числа N, що зберігається в К-розрядній комірці, необхідно:

1) перевести число N у двійкову систему числення;

2) отриманий результат доповнити зліва незначущими нулями до К розрядів.

Приклад 1

Отримати внутрішнє подання цілого числа 1607 у двобайтовій комірці.

Розв’язання

$$N = 1607 = 110\ 010\ 001\ 11_2.$$

Відповідь

Внутрішнє подання цього числа буде 0000 0110 0100 0111. Шістнадцяткова форма внутрішнього подання цього числа буде 0647.

Для подання цілого негативного числа використовується доповняльний код.

Доповняльним кодом двійкового числа X у N-розрядній комірці є число, яке доповнює його до значення 2^N.

Отримання доповняльного коду:

1) отримати внутрішнє подання позитивного числа N (прямий код);

2) отримати обернений код цього числа заміною 0 на 1 або 1 на 0;

3) до отриманого числа додати 1.

Позитивне число в прямому, оберненому і доповняльному кодах не змінює своє подання.

Використання доповняльного коду дозволяє замінити операцію віднімання операцією додавання, тобто процесору достатньо вміти лише додавати числа:

$$A-B = A + (- B).$$

Старший (К-й) розряд у внутрішньому поданні будь-якого позитивного числа дорівнює 0, негативного числа – 1. Тому цей розряд називається знаковим розрядом.

Приклад 2

Отримати внутрішнє подання цілого негативного числа -1607.

Розв’язання

1. Внутрішнє подання позитивного числа (прямий код) є таким:

000 0110 0100 0111.

2. Обернений код: 1111 1001 1011 1000.

3. Доповняльний код: 1111 1001 1011 1001.

Відповідь

Внутрішнє двійкове подання числа є таким: 1111 1001 1011 1001.

Дійсні числа подаються в ЕОМ у формі з плаваючою точкою. Цей формат використовує подання дійсного числа *R* у вигляді добутку мантиси *m* на основу системи числення *p* у деякому цілому степені *n*, який називають порядком:

$$R = m * p^n.$$

Подання числа в формі з плаваючою точкою є неоднозначним. Наприклад: $25.324 = 25324 * 10^1 = 0.0025324 * 10^4 = 2532.4 * 10^{-2}$.

У ЕОМ використовують нормалізоване подання числа у формі з плаваючою точкою. Мантиса в нормалізованому поданні повинна задовольняти умову

$$0.1_p < m < 1_p.$$

Тобто мантиса має бути менше одиниці, а перша значуща цифра – не нуль.

У пам'яті комп'ютера мантиса подається як ціле число, що містить тільки значущі цифри (0 цілих і кома не зберігається). Отже, внутрішнє подання дійсного числа зводиться до подання пари цілих чисел: мантиси і порядку.

Наприклад, у чотирибайтовій комірці пам'яті має міститися інформація про знак числа, порядок і значущі цифри мантиси. Розподіл цієї інформації за байтами буде таким:

Порядок	*Мантиса*		
1-й байт	2-й байт	3-й байт	4-й байт

У старшому біті старшого байта зберігається знак числа: 0 – плюс, 1 – мінус. Решта 7 бітів першого байта містять машинний порядок. У наступних трьох байтах зберігаються значущі цифри мантиси (24 розряди).

У семи двійкових розрядах поміщуються двійкові числа в діапазоні від 0000000 до 1111111. Значить, машинний порядок змінюється в діапазоні від 0 до 127 (у десятковій системі числення). Усього 128 значень. Порядок, очевидно, може бути як позитивним, так і негативним. Розумно ці 128 значень розділити порівну між позитивними і негативними значеннями порядку: від -64 до 63.

Машинний порядок зсунутий відносно математичного і має тільки позитивні значення. Зсув вибирається так, щоб мінімальному математичному значенню порядку відповідав нуль.

Зв'язок між машинним порядком (Мр) і математичним (р) у цьому випадку виражається формулою

$$Мр = р + 64.$$

Отримана формула записана в десятковій системі. У двійковій системі формула має вигляд

$$Мр_2 = р_2 + 1000000_2.$$

Для запису внутрішнього подання дійсного числа необхідно:

1) перевести модуль даного числа в двійкову систему числення з 24 значущими цифрами;

2) нормалізувати двійкове число;

3) знайти машинний порядок у двійковій системі числення;

4) з огляду на знак числа, записати його подання до чотирибайтового машинного слова.

Приклад 3

Записати внутрішнє подання числа 250,1875 у формі з плаваючою точкою.

Розв’язання

1. Запишемо число в двійковій системі числення з 24 значущими цифрами:

$$250{,}1875_{10} = 11111010,\ 0011000000000000000_2.$$

2. Запишемо нормалізоване подання двійкового числа в формі з плаваючою точкою:

$$0{,}11111010001100000000000*10_2^{1000}.$$

Тут мантиса, основа системи числення ($2_{10} = 10_2$) і порядок ($8_{10} = 1000_2$) записані в двійковій системі.

3. Обчислимо машинний порядок у двійковій системі числення:

$$Мр_2 = = 1000 + 100\ 0000 = 100\ 1000.$$

4. Запишемо подання числа в чотирибайтову комірку пам'яті з урахуванням знака числа:

0	1001000	11111010	00110000	00000000
31	30 24	23 16	15 8	7 0

Шістнадцяткова форма запису така: 48FA3000.

Приклад 4

За шістнадцятковою формою внутрішнього подання числа в формі з плаваючою точкою С9811000 відновити саме число.

Розв’язання

1. Перейдемо до двійкового подання числа в чотирибайтовій комірці, замінивши кожну шістнадцятирічну цифру чотирма двійковими цифрами:

1	1001001	10000001	00010000	00000000	.
31	24	23 16	15	7 0	

2. Зауважимо, що отриманий код є кодом негативного числа, оскільки в старшому розряді з номером 31 записана 1. Отримаємо порядок числа:

$$p=1001001_2 - 1000000_2 = 1001_2 = 9_{10}.$$

3. Запишемо нормалізовану форму двійкового числа з плаваючою точкою з урахуванням знака числа:

$$-0{,}10000001000100000000000 * 2^{1001}.$$

4. Число в двійковій системі числення має вигляд $-100000010{,}001_2$.

5. Переведемо число в десяткову систему числення:

$$-100000010{,}001_2 = -(1*2^8+1*2^1+1*2^{-3}) = -258{,}125_{10}.$$

Варіанти завдань

Запрограмувати виконання таких операцій і навести приклади виконання завдань без використання програмного забезпечення:

1. Подати чотири цілих числа з трьома розрядами у форматі з фіксованою точкою (комою) і чотири дробових числа (три розряди до коми та три після неї) у форматі з плаваючою точкою (комою).

2. Побудувати форму внутрішнього подання чисел:

– отримати двійкову форму внутрішнього подання цілого числа у вигляді двох байтів;

– отримати шістнадцяткову форму внутрішнього подання цілого числа у вигляді двох байтів;

– за шістнадцятковою формою внутрішнього подання цілого числа у вигляді двох байтів відновити саме число;

– отримати шістнадцяткову форму внутрішнього подання числа у форматі з плаваючою точкою у вигляді чотирьох байтів;

– за шістнадцятковою формою внутрішнього подання дійсного числа у вигляді чотирьох байтів відновити саме число.

3. Перевести чотири двійкових числа у строки десяткових цифрових символів.

4. Побудувати форму внутрішнього подання зображення у пам’яті.

Лабораторна робота № 3

ВИКОНАННЯ ПОРОЗРЯДНИХ АРИФМЕТИЧНИХ І ЛОГІЧНИХ ОПЕРАЦІЙ

Мета роботи: ознайомитися з принципами виконання порозрядних логічних операцій not, or, and, xor та арифметичних операцій +, -, *, /; навчитися виконувати двійкові арифметичні операції з перенесенням, відніманням за допомогою доповняльного коду.

Завдання

1. Виконати розрядні логічні операції між парами двійкових кодів.

2. За допомогою операцій not+1 побудувати доповняльний двійковий код позитивних чисел з перевіркою істинності.

3. Виконати двійкові арифметичні операції над парами цілих чисел.

4. Скласти звіт, який повинен містити тему, мету роботи, завдання, перелік правил для виконання операції, результати виконання завдань без використання програми, програмний код, результати виконання програми.

Теоретичні відомості

Логічне заперечення, або інверсія (not) виконується відповідно до таблиці істинності

A	неA
1	0
0	1

Логічне множення, або кон'юнкція (and) і логічне додавання, або диз'юнкція (or) виконуються відповідно до таблиці істинності

A	B	A and B	A or B
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Логічне віднімання, або виключне «або» (xor) виконується відповідно до таблиці істинності

A	B	A xor B
1	1	0
1	0	1
0	1	1
0	0	0

Для виконання арифметичних операцій в системі числення з основою *P* необхідно мати відповідні таблиці додавання і множення. Таблиці для *P* = 2, 8 і 16 подано нижче:

P =2

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

P =8

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

P =16

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

×	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Приклад 1

Виконати додавання чисел:

а – $10000000100_{(2)}$ + $111000010_{(2)}$;
б – $223,2_{(8)}$ + $427,54_{(8)}$;
в – $3B3,6_{(16)}$ + $38B,4_{(16)}$.

Розв’язання

а	10000000100	б	223,2	в	3B3,6
	+ 111000010		+ 427,54		+38B,4
	10111000110		652,74		73E,A

Відповідь

а – $10000000100_{(2)}$ + $111000010_{(2)}$ = $10111000110_{(2)}$;
б – $223,2_{(8)}$ + $427,54_{(8)}$ = $652,74_{(8)}$;
в – $3B3,6_{(16)}$ + $38B,4_{(16)}$ = $73E,A_{(16)}$.

Приклад 2

Виконати віднімання чисел:

а – $1100000011,011_{(2)}$ - $101010111,1_{(2)}$;
б – $1510,2_{(8)}$ - $1230,54_{(8)}$;
в – $27D,D8_{(16)}$ - $191,2_{(16)}$.

Розв’язання

а	1100000011,011	б	1510,2	в	27D,D8
	- 101010111,1		-1230,54		-191,2
	110101011,111		257,44		EC,B8

Відповідь

а – $1100000011,011_{(2)}$ - $101010111,1_{(2)}$ = $110101011,111_{(2)}$;
б – $1510,2_{(8)}$ - $1230,54_{(8)}$ = $257,44_{(8)}$;
в – $27D,D8_{(16)}$ - $191,2_{(16)}$ = $EC,B8_{(16)}$.

Приклад 3

Виконати множення чисел:

а – $100111_{(2)}$ x $1000111_{(2)}$;
б – $1170,64_{(8)}$ x $46,3_{(8)}$;
в – $61,A_{(16)}$ x $40,D_{(16)}$.

Розв’язання

	100111		1170,64		61,A
	* 1000111		* 46,3		*40,D
	100111		355 234		4F 52
а	+ 100111	б	+ 7324 70	в	+ 1868
	100111		47432 0		18B7,52
	100111		57334,134		
	101011010001				

Відповідь

а – $100111_{(2)}$ x $1000111_{(2)}$ = $101011010001_{(2)}$;
б – $1170,64_{(8)}$ x $46,3_{(8)}$ = $57334,134_{(8)}$;
в – $61,A_{(16)}$ x $40,D_{(16)}$ = $18B7,52_{(16)}$.

Варіанти завдань

Запрограмувати виконання операцій і навести приклади виконання завдань без використання програмного забезпечення:

– виконати розрядні логічні операції or, and, not or, not and між чотирма парами 16-розрядних двійкових кодів;

– за допомогою операцій not+1 побудувати 16-розрядний доповняльний двійковий код десяти позитивних чисел з перевіркою істинності;

– виконати двійкові арифметичних операції (+, -, *, /) над чотирма парами 16-розрядних цілих чисел.

Лабораторна робота № 4

ЛОГІЧНІ БІТОВІ ОПЕРАЦІЇ. ВИКОРИСТАННЯ МАСОК ДЛЯ ЗМІНИ БІТІВ ЧИСЛА

Мета роботи: ознайомитися і засвоїти роботу з командами бітових логічних операцій мови Assembler: ***NOT***, ***OR***, ***AND***, ***XOR***; ознайомитися з принципами використання масок для змін бітів числа, що знаходиться в реґістрі процесора.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.

2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.

3. Написати програму мовою Assembler, що виконує поставлене завдання. Результати обчислень вивести на екран в десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler до програми, написаної мовою високого рівня.

4. Протестувати програму на різних вхідних даних.

5. Скласти звіт, який повинен містити тему, мету роботи, завдання, опис вийняткових ситуацій та особливостей виконання завдання, лістинг програмного коду.

Теоретичні відомості

Розглянемо команди мови Assembler.

Команда пересилання даних копіює вміст джерела в приймач, джерело не змінюється:

MOV приймач, джерело.

Наприклад, команда ***MOV AX, 1*** присвоює реґістру ***AX*** значення *1*. Команда ***MOV AX, WORD PTR EAX*** записує в ***AX*** слово, яке знаходиться за адресою ***EAX***. Байт за адресою ***EAX*** записується в молодшу половину ***AX*** (в ***AL***), а байт за адресою ***EAX + 1*** – в ***AH*** (за законом Intel). Але не обов'язково, записуючи команди, використовувати таку складну адресацію. Наприклад, якщо є змінна ***y*** типу longint, то за допомогою команди ***MOV y, 10000*** їй можна присвоїти значення 10000. Можна також записати команду ***MOV DWORD PTR y, 10000***, показуючи, що ***y*** – 32-розрядна змінна. Якщо записати команду ***MOV DWORD PTR [y + 10], 5000***, то, починаючи з адреси ***[y + 10]***, в пам'ять буде записано 32-бітне число 5000. Додавання константи до адреси цілком коректне, оскільки всі вирази обчислюються на стадії компіляції і на швидкість програми не впливають. Нехай, наприклад, змінна ***y*** знаходиться за адресою 34567. Якщо записати ***[y + 10]***, то компілятор просто зрозуміє це як [32577], оскільки він знає адресу змінної ***y***. Операнди команди ***MOV*** можуть бути як реґістрами, так і змінними, але одночасно обидва операнди не можуть бути змінними.

Команда

XCHG операнд1, операнд2

обмінює операнди. Наприклад, якщо ***AL*** = 45, ***AH*** = 37, то після виконання ***XCHG AL, AH*** отримаємо ***AL*** = 37, ***AH*** = 45.

Команда

AND/OR приймач, джерело

виконує логічне побітне ***AND***/***OR*** над приймачем і джерелом і поміщує результат у приймач. Часто використовується для вибіркового встановлення в 0 або 1 окремих бітів.

Наприклад, команда ***AND AL, 00001111b*** обнулить старші чотири біти реґістра ***AL***, а молодші не змінить.

Команда

XOR приймач, джерело

реалізує логічне виключне АБО. Виконується побітове логічне виключне АБО над приймачем і джерелом, результат заноситься до приймача. Часто використовується для обнулення реґістрів.

Наприклад, ***XOR AX, AX*** обнуляє реґістр ***AX***, і робить це швидше, ніж ***MOV AX, 0***. Цією командою слід користуватися для обнулення реґістрів. Вона буде ефективно працювати на будь-якому Intel-сумісному комп'ютері. Ця команда офіційно підтримується Intel як каманда обнулення реґістра.

Команда

NOT приймач

кожен біт приймача, що дорівнює нулю, встановлює в 1 і кожен біт, що дорівнює 1, – в 0. Прапори не змінюються.

Приклад виконання роботи на C ++ з використанням вставок мовою Assembler

Завдання. Номер вказується в реґістрі ***EAX***. Установіть біти 0, 3, 5 в значення 1. Запишіть результат у реґістр ***EBX***. Вважати, що число додатне.

Розв’язання.

Застосування маски до числа називається маскуванням. Підбираючи маску і логічну операцію, можливо встановлювати певні біти числа в 0 або 1.

Розмір реґістра ***EAX*** становить 32 біти:

Число	Двійкове число	Десяткове число
	+00000000000000000000100000000	256
mask	+00000000000000000000000101001	41
Результат	+00000000000000000000100101001	297

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
#include <conio.h>

using namespace std;

void main()
```

```
{
	// creating the necessary variables
	int a; // declaration variable "a"
	int mask; // declaration variable "mask", with which we change the number
	int res; // declaration variable "res", to store the result

	// read variable values from file
	ifstream file_in; // creating object for input from file
	file_in.open("filein.txt"); // open file "filein.txt" for reading values "a" and "Mask"
	file_in >> a; // read variable "a"
	file_in >> mask; // variable "mask"
	file_in.close(); // close the file after reading the variables

	// using Assembler language
	__asm { // the beginning of the assembler insert
		mov ebx, a; // put in the register ebx the value of the variable "a"
		mov eax, mask; // put in the register eax the value of the variable "mask"
		or ebx, eax; // logical operation between the register ebx and the register eax.
Result is stored in the register ebx
		mov res, ebx; // assign the variable "res" result of a logical operation
	} // end of assembler insert

	// preparation of results
	char chA[256]; // array for variable "a" in binary code
	char chMask[256]; // array for variable "mask" in binary code
	char chRes[256]; // array for variable "res" in binary code
	_itoa_s(a, chA, 2); // converting the variable "a" to binary code using the itoa_s
functionand saving it to the chA array
	_itoa_s(mask, chMask, 2); // converting the variable "a" to binary code using the itoa_s
functionand saving it to the chMask array
	_itoa_s(res, chRes, 2); // converting the variable "a" to binary code using the _itoa_s
functionand saving it to the chRes array

	// output to the file
	ofstream file_out; // creating object for write results to file
	file_out.open("fileout.txt"); // opening file "fileout.txt" to record results
	file_out << "A =" << a << endl; // record to file value the variable "a"
	file_out << "Mask =" << mask << endl; // record to file value the variable "Mask"
	file_out << "Res =" << res << endl; // record to file value the variable "Res"
	file_out << endl; // empty string
	file_out << "In binary code:" << endl; // record to file phrases "In binary code: "
	file_out << endl; // empty string
	file_out << "A =" << chA << endl; // record to file value the variable "a" in binary
code
	file_out << "Mask =" << chMask << endl; // record to file value the variable "Mask" in
binary code
	file_out << "Res =" << chRes << endl; // record to file value the variable "Res" in
binary code
	file_out.close(); // close the file after record the variables

	// output to console
	cout << "A =" << a << endl; // record to console value the variable "a"
	cout << "Mask =" << mask << endl; // record to console value the variable "Mask"
	cout << "Res =" << res << endl; // record to console value the variable "Res"
	cout << endl; // empty string
	cout << "In binary code:" << endl; // record to console phrases "In binary code: "
	cout << endl; // empty string
	cout << "A =" << chA << endl; // record to file console the variable "a" in binary code
	cout << "Mask =" << chMask << endl; // record to console value the variable "Mask" in
binary code
	cout << "Res =" << chRes << endl; // record to file console the variable "res" in binary
code

	system("pause"); // waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 4.1, а результат виконання програми – на рисунках 4.2 і 4.3.

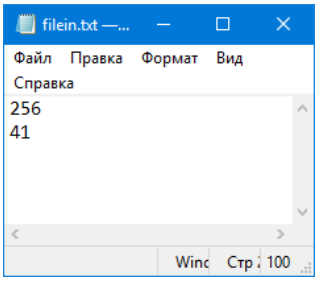


Рисунок 4.1 – Файл із початковими номерами та масками

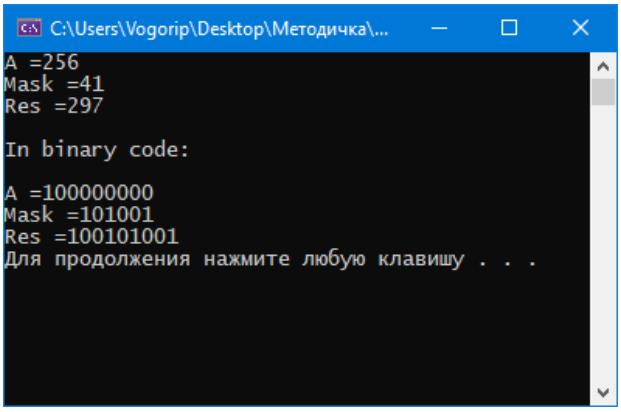


Рисунок 4.2 – Результат у консолі

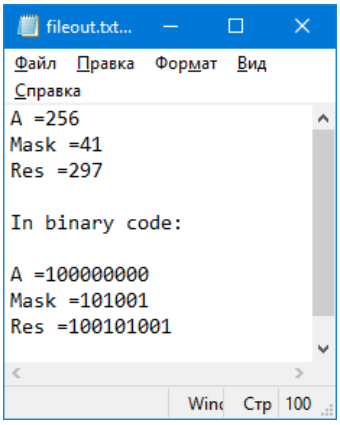


Рисунок 4.3 – Результат у файлі

Варіанти завдань

Завдання 1. Число задано у регістрі. Без використання операцій умовного переходу встановити відповідні біти в значення 0 або 1. Результат записати в інший регістр. Варіанти завдань наведено у таблиці 4.1.

Завдання 2. Без використання операцій умовного переходу встановити регістр числа таким, що дорівнює полю заданого регістра від *n* до *m* бітів, інші біти встановити в 0. Уважати, що число позитивне/негативне. Знак числа повинен залишитися незмінним. Варіанти завдань наведено у таблиці 4.2.

Таблиця 4.1 – Варіанти завдання 1

Номер варіанта	Число задано у реґістрі	Номери бітів	Значення бітів	Реґістр результату
1	AL	0, 5, 7	0	BH
2	BL	1, 5, 7	0	BH
3	CL	2, 5, 7	0	BH
4	DL	3, 5, 7	0	BH
5	AH	0, 4, 6	0	BH
6	CH	1, 4, 6	0	BH
7	DH	2, 4, 6	0	BH
8	AL	0, 4, 6	1	BL
9	CL	1, 4, 6	1	BL
10	DL	2, 4, 6	1	BL
11	AH	0, 5, 7	1	BL
12	BH	1, 5, 7	1	BL
13	CH	2, 5, 7	1	BL
14	DH	3, 5, 7	1	BL
15	AX	7, 11, 13	0	BX
16	CX	8, 11, 13	0	BX
17	DX	9, 11, 13	0	BX
18	AX	7, 11, 13	1	BX
19	CX	8, 11, 13	1	BX
20	DX	9, 11, 13	1	BX
21	EAX	11, 17, 27	0	EBX
22	ECX	13, 17, 27	0	EBX
23	EDX	15, 17, 27	0	EBX
24	EAX	11, 17, 27	1	EBX
25	ECX	13, 17, 27	1	EBX
26	EDX	15, 17, 27	1	EBX
27	AL	0, 3, 7	0	BH
28	BL	1, 3, 7	0	BH
29	CL	2, 3, 7	0	BH
30	DL	3, 4, 7	0	BH
31	AH	0, 4, 7	0	BH
32	CH	1, 4, 7	0	BH
33	DH	2, 4, 7	0	BH
34	AL	0, 4, 7	1	BL
35	CL	1, 4, 7	1	BL
36	DL	2, 4, 7	1	BL
37	AH	0, 3, 7	1	BL
38	BH	1, 3, 7	1	BL
39	CH	2, 3, 7	1	BL
40	DH	3, 4, 7	1	BL

Продовження таблиці 4.1

Номер варіанта	Число задано у реґістрі	Номери бітів	Значення бітів	Реґістр результату
41	AX	7, 13, 15	0	BX
42	CX	8, 13, 15	0	BX
43	DX	9, 13, 15	0	BX
44	AX	7, 13, 15	1	BX
45	CX	8, 13, 15	1	BX
46	DX	9, 13, 15	1	BX
47	EAX	11, 21, 31	0	EBX
48	ECX	13, 21, 31	0	EBX
49	EDX	15, 21, 31	0	EBX
50	EAX	11, 21, 31	1	EBX

Таблиця 4.2 – Варіанти завдання 2

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Поле реґістра [*n*..*m*]	Реґістр результату
1	AH	1	0..5	BL
2	BH	1	1..5	BL
3	CH	1	2..5	BL
4	DH	1	3..5	BL
5	AL	1	0..6	BL
6	CL	1	1..6	BL
7	DL	1	2..6	BL
8	AH	0	0..6	BH
9	CH	0	1..6	BH
10	DH	0	2..6	BH
11	AL	0	0..5	BH
12	BL	0	1..5	BH
13	CL	0	2..5	BH
14	DL	0	3..5	BH
15	BX	1	7..13	AX
16	CX	1	8..13	AX
17	DX	1	9..13	AX
18	BX	0	7..13	AX
19	CX	0	8..13	AX
20	DX	0	9..13	AX
21	EBX	1	11..23	EAX
22	ECX	1	13..23	EAX
23	EDX	1	15..23	EAX
24	EBX	0	11..23	EAX
25	ECX	0	13..23	EAX

Продовження таблиці 4.2

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Поле реґістра [*n*..*m*]	Реґістр результату
26	EDX	0	15..23	EAX
27	AH	1	1..5	BL
28	BH	1	2..5	BL
29	CH	1	3..5	BL
30	DH	1	4..5	BL
31	AL	1	1..6	BL
32	CL	1	2..6	BL
33	DL	1	3..6	BL
34	AH	0	1..6	BH
35	CH	0	2..6	BH
36	DH	0	3..6	BH
37	AL	0	1..5	BH
38	BL	0	2..5	BH
39	CL	0	3..5	BH
40	DL	0	4..5	BH
41	BX	1	5..11	AX
42	CX	1	6..11	AX
43	DX	1	7..11	AX
44	BX	0	5..11	AX
45	CX	0	6..11	AX
46	DX	0	7..11	AX
47	EBX	1	15..25	EAX
48	ECX	1	17..25	EAX
49	EDX	1	19..25	EAX
50	EBX	0	15..25	EAX

Лабораторна робота № 5

ОПЕРАЦІЇ НАД БІТАМИ (ПОШУК, ІНВЕРСІЯ, СКИДАННЯ АБО УСТАНОВКА БІТА)

Мета роботи: засвоїти принципи роботи з бітами в реґістрах процесора мовою Assembler за допомогою команд ***BT***, ***BTS***, ***BTR***, ***BTC***, ***BSF***, ***BSR***.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.

2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.

3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран в десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.

4. Протестувати програму на різноманітних вхідних даних.

5. Скласти звіт, який повинен містити тему і мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістинг програмного коду.

Теоретичні відомості

Розглянемо команди мови Assembler.

Команда ***BT база, зсув*** – перевірка біта. Зчитує в прапор ***CF*** значення біта з бітового рядка, визначеного першим операндом – бітовою базою (реґістр/змінна). Зсув – число або реґістр. Якщо це реґістр, то розрядність його повинна збігатися з розрядністю бази. Розрядність бітового рядка – 16/32. Коли перший операнд – реґістр, зсув не може перевищувати 15/31 (залежно від розміру реґістра). Якщо перевищує, то береться залишок від ділення на 16/32. Якщо перший операнд – змінна, то як бітова база потрібен біт 0 зазначеного байта в пам'яті, а зсув від 0 до 31. Наприклад, x = 0000000001100100b. Тоді після виконання команди ***BT WORD PTR x, 0*** буде ***CF*** = 0 (останній біт), після ***BT WORD PTR x, 1*** буде ***CF*** = 0 (передостанній біт), після ***BT WORD PTR x, 2*** буде ***CF*** = 1 (передпередостанній біт). Не слід користуватися командою ***BT*** поблизу недоступних для читання областей пам'яті!

Після виконання команди ***BT*** прапори ***OF***, ***SF***, ***ZF***, ***AF***, ***PF*** не визначені. Наприклад, після виконання послідовності ***MOV AX, 1; BT AX, 0;*** прапор ***CF*** = 1 (нульовому біту). Після виконання послідовності ***MOV AX, 1; BT AX, 1;*** прапор ***CF*** = 0 (першому біту).

Команда ***BTS база, зсув*** – перевірка й установлення біта.

Команда ***BTR база, зсув*** – перевірка й скидання біта.

Команда ***BTC база, зсув*** – перевірка й інверсія біта.

Ці команди діють аналогічно ***BT***. Але вони не тільки перевіряють біт (установлюють ***CF***), але і змінюють його. Після виконання команд ***BTS/BTR/BTC*** прапор ***CF*** дорівнює значенню біта до його зміни, інші прапори не визначені.

Команда ***BSF приймач, джерело*** – прямий пошук біта.

Команда ***BSR приймач, джерело*** – зворотний пошук біта.

Приймач – це реґістр, джерело може бути реґістром чи змінною. Команда ***BSF*** сканує джерело (r/m) розрядністю 16/32 починаючи з молодшого біта і записує в приймач (r16/r32) номер першого зустрінутого біта, що дорівнює 1. Команда ***BSR*** сканує джерело починаючи зі старшого біта. Тобто якщо джерело дорівнює 0000000000001010b, то ***BSF*** поверне 1, а ***BSR*** поверне 3. Якщо джерело дорівнює нулю, значення приймача не визначене, і ***ZF*** = 1, інакше ***ZF*** = 0. Прапори ***CF***, ***OF***, ***SF***, ***AF***, ***PF*** не визначені.

Команда ***SETCC приймач*** – установлення байтів за умовою. Це набір команд, що встановлюють приймач (r8/m8) у 1 або 0, якщо задовольняється або не задовольняється певна умова.

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання 1. У реґістрі ***CL*** число встановлюється випадковим чином. Визначити номер першого зустрінутого біта, що дорівнює 1. Число додатне.

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
#include <conio.h>
#include <windows.h>

using namespace std;

int main()
{
	//creating the necessary variables
	char c;// = char(56);
	short res;
	//read variable values from file
	ifstream file_in;			//creating object for input from file
	file_in.open("filein.txt"); //open file "filein.txt" for reading values "a" and "mask"
	file_in >> c;				//read variable "a"
	file_in.close();			//close the file after reading the variables
	//using Assembler language
	_asm {
		bsf ax, c;//search for the first bit equal to 1
		mov res, ax;//assign the variable "res" result of a logical operation
	}					//end of assembler insert
	//preparation of results
	int c1 = (int)c;//convert character to number
	char ch_c1[33] = "";//array for variable "a" in binary code
	_itoa_s(c1, ch_c1, 2);//converting the variable "a" to binary code using the _itoa_s
function and saving it to the ch_a1 array
					//output to the file
	ofstream file_out;			// creating object for write results to file
	file_out.open("fileout.txt");		//opening file "fileout.txt" to record
results
	file_out << "a=    " << c << endl;	//record to file value the variable "a"
	file_out << endl;			//empty string
	file_out << "In binary code:" << endl;	//record  to file phrases "In binary code:"
	file_out << endl;			//empty string
	file_out << "a=    " << ch_c1 << endl;	  //record to file value the variable "a" in
binary code
```

```
	int res1 = (int)res;//convert character to number
	file_out << "res=  " << res1 << endl;		//record to file value the variable "res"
	file_out.close();				//close the file after record the variables
	                                                        //output to
console
	cout << "a=    " << c << endl;		//record to console value the variable "a""
	cout << "In binary code:" << endl;	//record  to console phrases "In binary code:"
	cout << "a=    " << ch_c1 << endl;	  //record to file console the variable "a" in
binary code
	cout << "res=  " << res1 << endl;	  //record to console value the variable "res"

	system("pause"); //waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 5.1, а результат виконання програми – на рисунках 5.2 і 5.3.

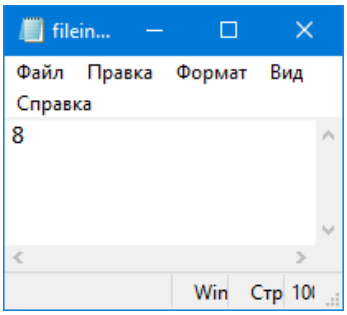


Рисунок 5.1 – Вхідний файл

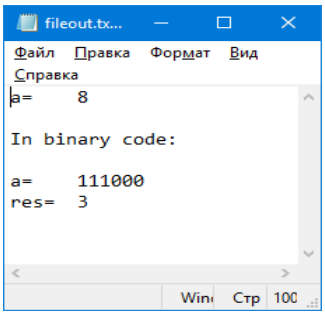


Рисунок 5.2 – Файл результату

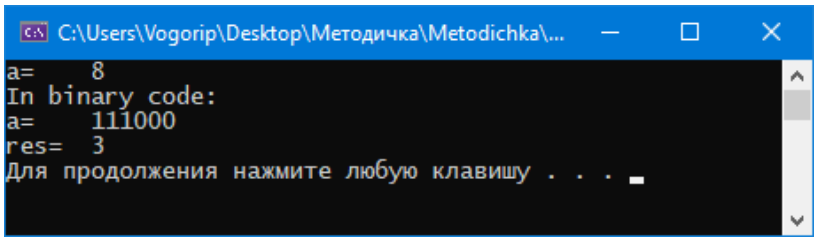


Рисунок 5.3 – Результат програми в консолі

Завдання 2. У регістрі ***EAX*** число встановлюється випадковим чином. Установити перші 15 бітів у значення *1*.

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
#include <conio.h>
#include <windows.h>

using namespace std;

int main()
{
	//creating the necessary variables
	int a;
	int b = 15;
	int res = 0;
	//read variable values from file
	ifstream file_in;			//creating object for input from file
	file_in.open("filein.txt"); //open file "filein.txt" for reading values "a" and "mask"
```

```
	file_in >> a;				//read variable "a"
	file_in.close();			//close the file after reading the variables
	char ch_a1[256] = "";//array for variable "a" in binary code
	_itoa_s(a, ch_a1, 2);//converting the variable "a" to binary code using the _itoa_s
function and saving it to the ch_a1 array
	                                    //output to the file
	cout << "a=    " << a << endl;		//record to console value the variable "a""
	cout << "In binary code:" << endl;	//record  to console phrases "In binary code:"
	cout << "a=    " << ch_a1 << endl;	  //record to file console the variable "a" in
binary code
	ofstream file_out;				// creating object for write results to file
	file_out.open("fileout.txt");			//opening file "fileout.txt" to record
results
	file_out << "a=    " << a << endl;		//record to file value the variable "a"
	file_out << endl;
	file_out << "In binary code:" << endl;	//record  to file phrases "In binary code:"
	file_out << endl;				//empty string
	file_out << "a=    " << ch_a1 << endl;	  //record to file value the variable "a" in
binary code
	_asm
	{
		mov eax, a;//put in the register eax the value of the variable "a"
		mov ebx, b;//put in the register ebx the value of the variable "b"
		BTS eAX, ebx;// replace 15 bits with 1
		mov res, eax;//assign the variable "res" result of a logical operation
	}			//end of assembler insert
	file_out << "res=    " << res << endl;		//record to file value the variable "res"
	file_out << endl;
	char ch_res[256] = "";//array for variable "res" in binary code
	_itoa_s(res, ch_res, 2);//converting the variable "res" to binary code using the _itoa_s
function and saving it to the ch_res array
	                                  //output to the file
	file_out << "In binary code:" << endl;	//record  to file phrases "In binary code:"
	file_out << endl;				//empty string
	file_out << "res=    " << ch_res << endl;	//record to file value the variable
"res" in binary code
	cout << "res=    " << res << endl;		//record to console value the variable "res"
	cout << "In binary code:" << endl;	//record  to console phrases "In binary code:"
	cout << "res=    " << ch_res << endl;	  //record to file console the variable "res"
in binary code
	system("pause");//waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 5.4, а результат виконання програми – на рисунках 5.5 і 5.6.

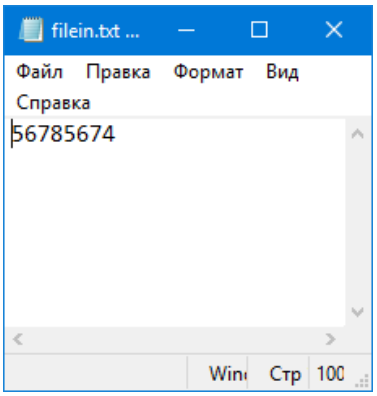


Рисунок 5.4 – Вхідний файл

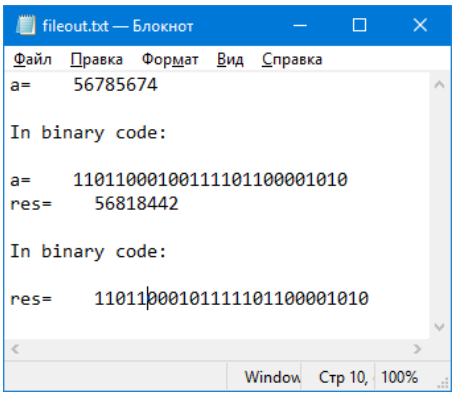


Рисунок 5.5 – Файл з результатами

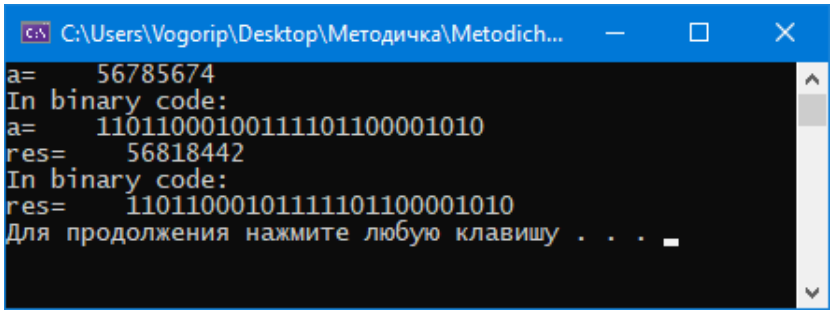


Рисунок 5.6 – Результат програми на консолі

Варіанти завдань

Завдання 1. У регістрі задано число випадковим чином. Без використання операцій умовного переходу визначити номер першого зустрінутого біта, що дорівнює 0 або 1. Число позитивне/негативне. Варіанти завдань наведено у таблиці 5.1.

Завдання 2. У регістрі задано число випадковим чином. Без використання операцій умовного переходу встановити *n*-й біт у значення 0 або 1. Варіанти завдань наведено у таблиці 5.2.

Таблиця 5.1 – Варіанти завдання 1

Номер варіанта	Число задано у регістрі	Число додатне (0)/ від'ємне (1)	Значення шуканого біта
1	EAX	0	0
2	EBX	0	0
3	ECX	0	0
4	EDX	0	0
5	EAX	1	0
6	EBX	1	0
7	ECX	1	0
8	EDX	1	0
9	EAX	0	1
10	EBX	0	1
11	ECX	0	1
12	EDX	0	1
13	EAX	1	1
14	EBX	1	1
15	ECX	1	1
16	EDX	1	1
17	AX	0	0
18	BX	0	0
19	CX	0	0
20	DX	0	0

Продовження таблиці 5.1

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від’ємне (1)	Значення шуканого біта
21	AX	1	0
22	BX	1	0
23	CX	1	0
24	DX	1	0
25	AX	0	1
26	BX	0	1
27	CX	0	1
28	DX	0	1
29	AX	1	1
30	BX	1	1
31	CX	1	1
32	DX	1	1
33	AL	0	0
34	BL	0	0
35	CL	0	0
36	DL	0	0
37	AL	1	0
38	BL	1	0
39	CL	1	0
40	DL	1	0
41	AL	0	1
42	BL	0	1
43	CL	0	1
44	DL	0	1
45	AL	1	1
46	BL	1	1
47	CL	1	1
48	DL	1	1

Таблиця 5.2 – Варіанти завдання 2

Номер варіанта	Число задано у реґістрі	Номер біта (*n*)	Встановити біт у значення
1	AX	0	0
2	BX	1	1
3	CX	2	0
4	DX	3	1
5	AX	4	0
6	BX	5	1
7	CX	6	0
8	DX	7	1

Продовження таблиці 5.2

Номер варіанта	Число задано у регістрі	Номер біта (*n*)	Встановити біт у значення
9	AX	8	0
10	BX	9	1
11	CX	10	0
12	DX	11	1
13	AX	12	0
14	BX	13	1
15	CX	14	0
16	DX	15	1
17	EAX	16	0
18	EBX	17	1
19	ECX	18	0
20	EDX	19	1
21	EAX	20	0
22	EBX	21	1
23	ECX	22	0
24	EDX	23	1
25	EAX	24	0
26	EBX	25	1
27	ECX	26	0
28	EDX	27	1
29	EAX	28	0
30	EBX	29	1
31	ECX	30	0
32	EDX	31	1
33	AL	0	0
34	BL	1	1
35	CL	2	0
36	DL	3	1
37	AL	4	0
38	BL	5	1
39	CL	6	0
40	DL	7	1
41	AH	0	0
42	BH	1	1
43	CH	2	0
44	DH	3	1
45	AH	4	0
46	BH	5	1
47	CH	6	0
48	DH	7	1

Лабораторна робота № 6

ОПЕРАЦІЇ ЗСУВУ

Мета роботи: вивчити принципи роботи зсувів у реґістрах процесора мовою Assembler за допомогою команд ***SHL***, ***SHR***, ***SAL***, ***SAR***, ***ROR***, ***ROL***, ***RCR***, ***RCL***,***SHRD***, ***SHLD***.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.
2. Проаналізувати завдання. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.
3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран у десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.
4. Протестувати програму на різних вхідних даних.
5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістінг програмного коду.

Теоретичні відомості

Команди ***SHL/SHR/SAL/SAR приймач, число/CL*** – це команди зсуву (рисунок 6.1). Якщо вказаний реґістр ***CL***, то у ньому враховуються тільки молодші п’ять бітів.

Команди ***SHL*** і ***SAL*** – арифметичний і логічний зсув, код цих команд однаковий.

Команда ***SAR*** відрізняється від ***SHR*** тим, що не обнуляє старший біт, тому майже еквівалентна знаковому діленню на 2, але на відміну від ***IDIV*** округлення відбувається не в бік нуля, а в бік негативної нескінченності.

Команди ***SHL***/***SAL*** переводять старший біт у ***CF***, а команди ***SHR/SAR*** переводять молодший біт у ***CF***. Зсуви на 1 змінюють значення ***OF: SHL/SAL*** установлюють його в 1, якщо старший біт змінився, інакше ***OF*** = 0. Команда ***SHR*** установлює ***OF*** у значення старшого біта вихідного числа, а ***SAR*** обнуляє ***OF***. Для зсувів на кілька бітів значення ***OF*** не визначене. ***ZF***, ***SF***, ***PF*** призначаються відповідно до результату; ***AF*** не визначений. Розрядність операндів – 8/16/32.

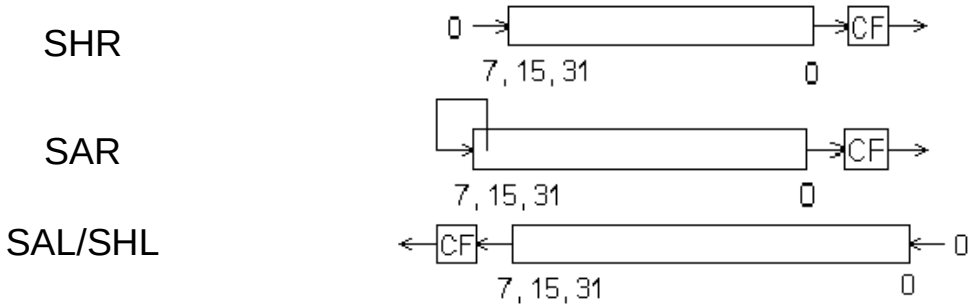


Рисунок 6.1 – Команди зсуву

Розглянемо команди циклічного зсуву (рисунок 6.2).

Команди ***ROR/ROL приймач, лічильник*** – циклічний зсув вправо/вліво.

Команди ***RCR/RCL приймач, лічильник*** – циклічний зсув вправо/вліво через прапор перенесення.

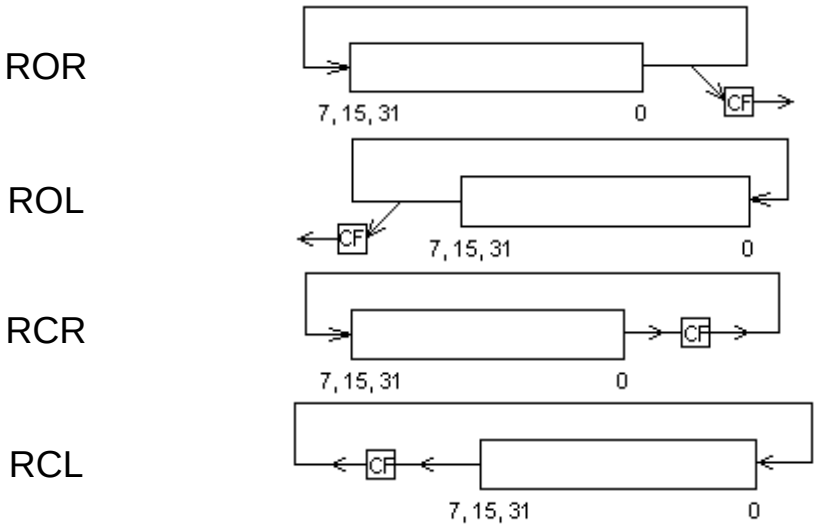


Рисунок 6.2 – Команди циклічного зсуву

Приймач – реґістр/змінна, лічильник – число/***CL***, з якого враховуються молодші п’ять бітів. При виконанні циклічного зсуву команди ***ROR/ROL*** переміщують кожний біт приймача вправо/вліво на одну позицію, за винятком молодшого/старшого, який записується в позицію старшого/молодшого біта.

Наприклад, якщо ***AH*** = 01100100b, то після виконання ***ROR AH, 4*** отримаємо ***AH*** = 01000110b. Після виконання команд циклічного зсуву прапор ***CF*** завжди дорівнює останньому біту, що вийшов за межі приймача.

Наприклад, після виконання послідовності ***MOV AL, 00001011b; ROR AL, 4;*** прапор ***CF*** дорівнюватиме 1 (що була третім бітом в ***AL***). Прапор ***OF*** визначений тільки для зсувів на 1 – він встановлюється, якщо змінювалося значення старшого біта, інакше він скидається. Прапори ***SF***, ***ZF***, ***AF***, ***PF*** не змінюються.

Команди ***SHRD/SHLD приймач, джерело, лічильник*** – зсув підвищеної точності вправо/вліво (рисунок 6.3). Приймач (реґістр/змінна) зсувається вправо/вліво на число бітів, вказане в лічильнику (число/***CL***). Якщо лічильник – реґістр ***CL***, то враховуються тільки молодші п'ять бітів. Старший (для ***SHLD***) або молодший (для ***SHRD***) біти НЕ обнуляються, а зчитуються з джерела (реґістру), значення якого не змінюється.

Наприклад, якщо приймач містить 00000000 01000101b (69), а джерело – 10101010 10101010b (43690), лічильник дорівнює 3, то результатом ***SHRD*** буде 01000000 00001000b (16392), а результатом ***SHLD*** – 00000010 00101101b (557). Розрядність приймача і джерела – 16/32.

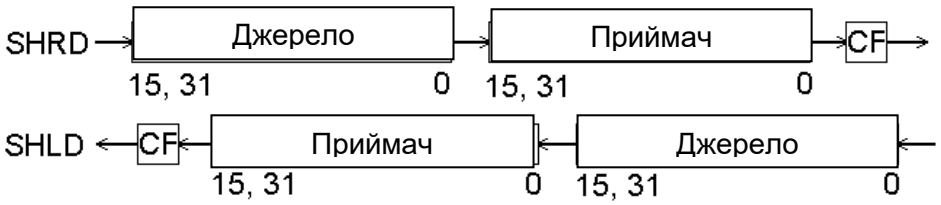


Рисунок 6.3 – Команди зсуву підвищеної точності

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання. Установити для ***BX*** число, яке дорівнює полю реґістра ***AX*** від 2 до 11 бітів. Число додатне. Розмір реґістра ***AX*** – 16 бітів.

Число	Двійкове число	Десяткове число
	0000000100000000	256
Результат	0000000001000000	64

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
#include <conio.h>
#include <windows.h>

using namespace std;

void main()
{
	//creating the necessary variables
	short int a;    //declaration variable "a"
	short int res;  //declaration variable "res", to store the result

	//read variable values from file
	ifstream file_in;           //creating object for input from file
	file_in.open("filein.txt"); //open file "filein.txt" for reading values "a" and
"mask"
	file_in >> a;               //read variable "a"
	file_in.close();            //close the file after reading the variables

	//using Assembler language
```

```
	__asm {				//the beginning of the assembler insert
		mov bx, a;		//put in the register ebx the value of the variable "a"
		shl bx, 4;		//4 bit left shift
		shr bx, 6;		//6 bit left
		mov res, bx;		//assign the variable "res" result of a logical operation
	}				//end of assembler insert

	//preparation of results
	char chA[256];			//array for variable "a" in binary code
	char chRes[256];		//array for variable "res" in binary code
	_itoa_s(a, chA, 2);		//converting the variable "a" to binary code using the
_itoa_s function and saving it to the chA array
	_itoa_s(res, chRes, 2);		//converting the variable "a" to binary code using the
_itoa_s function and saving it to the chRes array

	//output to the file
	ofstream file_out;				// creating object for write results to
file
	file_out.open("fileout.txt");			//opening file "fileout.txt" to record
results
	file_out << "a=    " << a << endl;		//record to file value the variable "a"
	file_out << "res=  " << res << endl;		//record to file value the variable
"res"
	file_out << endl;				//empty string
	file_out << "In binary code:" << endl;	//record  to file phrases "In binary
code:"
	file_out << endl;				//empty string
	file_out << "a=    " << chA << endl;		//record to file value the variable "a"
in binary code
	file_out << "res=  " << chRes << endl;		//record to file value the variable
"res" in binary code
	file_out.close();				//close the file after record the
variables

	//output to console
	cout << "a=    " << a << endl;		//record to console value the variable "a""
	cout << "res=  " << res << endl;		//record to console value the variable "res"
	cout << endl;				//empty string
	cout << "In binary code:" << endl;	//record  to console phrases "In binary
code:"
	cout << endl;				//empty string
	cout << "a=    " << chA << endl;		//record to file console the variable "a" in
binary code
	cout << "res=  " << chRes << endl;		//record to file console the variable "res"
in binary code

	system("pause"); //waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 6.4, а результат виконання програми – на рисунках 6.5 і 6.6.

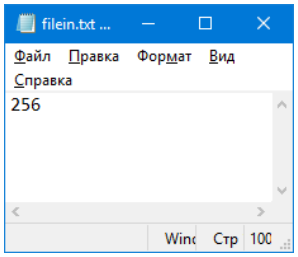


Рисунок 6.4 – Файл із початковими номерами та масками

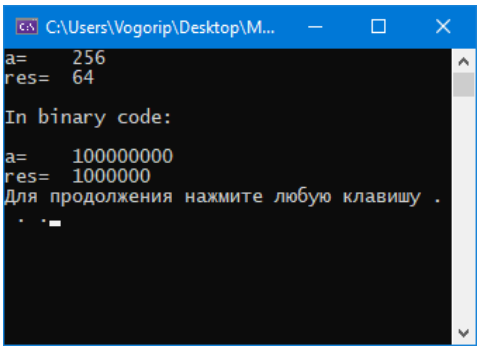


Рисунок 6.5 – Результат виконання у консолі

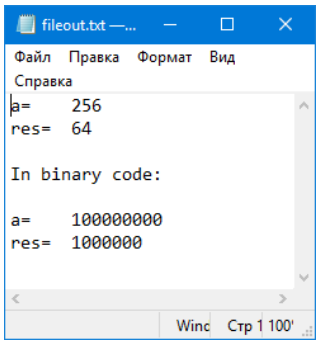


Рисунок 6.6 – Результат виконання у файлі

Варіанти завдань

Завдання 1. У реґістрі задано число. Без використання операцій умовного переходу встановити інший реґістр у значення 1, якщо *n*-й біт вихідного числа дорівнює 1, і у значення 0, якщо *n*-й біт дорівнює 0. Уважати, що число позитивне/негативне. Варіанти завдань наведено у таблиці 6.1.

Завдання 2. Без використання операцій умовного переходу встановити заданий реґістр таким, що дорівнює полю іншого реґістра від *n* до *m* бітів починаючи з молодшого біта. Знак числа повинен залишитися незмінним. Варіанти завдань наведено у таблиці 6.2.

Таблиця 6.1 – Варіанти завдання 1

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Номер біта (*n*)	Установити реґістр
1	AX	0	1	BX
2	CX	1	2	BX
3	DX	0	3	BX
4	AX	1	4	BX
5	CX	0	5	BX

Продовження таблиці 6.1

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від’ємне (1)	Номер біта (*n*)	Установити реґістр
6	DX	1	6	BX
7	AX	0	7	BX
8	CX	1	8	BX
9	DX	0	9	BX
10	AX	1	10	BX
11	CX	0	11	BX
12	DX	1	12	BX
13	DX	0	13	BX
14	AX	1	14	BX
15	CX	0	2	BX
16	DX	1	3	BX
17	AL	0	0	BH
18	BL	1	1	BH
19	CL	0	2	BH
20	DL	1	3	BH
21	AH	0	4	BH
22	CH	1	5	BH
23	DH	0	6	BH
24	AL	1	0	BL
25	CL	0	1	BL
26	DL	1	2	BL
27	AH	0	3	BL
28	BH	1	4	BL
29	CH	0	5	BL
30	DH	1	6	BL
31	EAX	0	16	EBX
32	ECX	1	17	EBX
33	EDX	0	18	EBX
34	EAX	1	19	EBX
35	ECX	0	20	EBX
36	EDX	1	21	EBX
37	EAX	0	22	EBX
38	ECX	1	23	EBX
39	EDX	0	24	EBX
40	EAX	1	25	EBX
41	ECX	0	26	EBX
42	EDX	1	27	EBX
43	EAX	0	28	EBX
44	ECX	1	29	EBX
45	EDX	0	30	EBX

Продовження таблиці 6.1

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Номер біта (*n*)	Установити реґістр
46	EBX	1	16	EAX
47	ECX	0	17	EAX
48	EDX	1	18	EAX
49	EBX	0	19	EAX
50	ECX	1	20	EAX

Таблиця 6.2 – Варіанти завдання 2

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Поле реґістра [*n..m*]	Реґістр для запису результату
1	BX	1	7..13	AX
2	CX	1	8..13	AX
3	DX	1	9..13	AX
4	BX	0	7..13	AX
5	CX	0	8..13	AX
6	DX	0	9..13	AX
7	EBX	1	11..23	EAX
8	ECX	1	13..23	EAX
9	EDX	1	15..23	EAX
10	EBX	0	11..23	EAX
11	ECX	0	13..23	EAX
12	EDX	0	15..23	EAX
13	AH	1	1..5	BL
14	BH	1	2..5	BL
15	CH	1	3..5	BL
16	DH	1	4..5	BL
17	AL	1	1..6	BL
18	CL	1	2..6	BL
19	DL	1	3..6	BL
20	AH	0	1..6	BH
21	CH	0	2..6	BH
22	DH	0	3..6	BH
23	AL	0	1..5	BH
24	BL	0	2..5	BH
25	CL	0	3..5	BH
26	DL	0	4..5	BH
27	AH	1	0..5	BL
28	BH	1	1..5	BL

Продовження таблиці 6.2

Номер варіанта	Число задано у реґістрі	Число додатне (0)/ від'ємне (1)	Поле реґістра [*n..m*]	Реґістр для запису результату
29	CH	1	2..5	BL
30	DH	1	3..5	BL
31	AL	1	0..6	BL
32	CL	1	1..6	BL
33	DL	1	2..6	BL
34	AH	0	0..6	BH
35	CH	0	1..6	BH
36	DH	0	2..6	BH
37	AL	0	0..5	BH
38	BL	0	1..5	BH
39	CL	0	2..5	BH
40	DL	0	3..5	BH
41	BX	1	5..11	AX
42	CX	1	6..11	AX
43	DX	1	7..11	AX
44	BX	0	5..11	AX
45	CX	0	6..11	AX
46	DX	0	7..11	AX
47	EBX	1	15..25	EAX
48	ECX	1	17..25	EAX
49	EDX	1	19..25	EAX
50	EBX	0	15..25	EAX

Лабораторна робота № 7

ОБЧИСЛЕННЯ АРИФМЕТИЧНИХ ВИРАЗІВ

Мета роботи: ознайомитися з принципами роботи арифметичних машинозалежних операцій мови Assembler за допомогою команд ***ADD***, ***ADC***, ***SUB***, ***SBB***, ***INC***, ***DEC***, ***MUL***, ***IMUL***, ***DIV***, ***IDIV***, а також команд зміни розміру операндів ***CBW***, ***CWD***, ***CWDE***, ***CDQ*** і команди зміни знака ***NEG***; вивчити застосування зсувів у реґістрах процесора для здійснення операції множення або ділення на число, що є степенем числа 2.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.

2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначених змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання..

3. Написати програму мовою Assembler, що виконує поставлене завдання. Результати обчислень вивести на екран в десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.

4. Протестувати програму на різних вхідних даних.

5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістінг програмного коду.

Теоретичні відомості

Команда ***ADD приймач, джерело*** виконує додавання приймача і джерела, результат заносить у приймач. Джерело не змінюється, але змінюються прапори.

Команда ***ADC приймач, джерело*** виконує додавання приймача, джерела і прапора ***CF***. Зазвичай ця команда використовується для складання чисел підвищеної точності.

Наприклад, є два 64-бітних числа: перше в ***EDX: EAX*** (молодше подвійне слово в ***EAX***, старше подвійне слово в ***EDX***), друге – в ***EBX: ECX***. Тоді після виконання команд ***ADD EAX, ECX; ADC EDX, EBX;*** у парі реґістрів ***EDX: EAX*** буде знаходитися сума цих 64-бітних чисел. Раніше 16-розрядні процесори подібним чином додавали 32-бітні числа (вважаючи, що кожне складається з двох 16-бітних). Зараз це не має сенсу. Більш того, процесору все одно, які числа додавати – 16-розрядні чи 32-розрядні, швидкість однакова. Тому складання безпосередньо 32-бітних чисел швидше, ніж обчислення з розбиттям і складанням 16-бітних (у два рази). Але іноді це може знадобитися (наприклад, Турбо-Паскаль не може обробляти 32-бітові реґістри).

Команда ***SUB приймач, джерело*** віднімає джерело від приймача, результат заносить в приймач.

Команда ***SBB приймач, джерело*** віднімає від приймача значення джерела, потім віднімає значення ***CF***. Її також можна використовувати для обчислення 64-бітних слів.

Команда ***INC приймач*** виконує такі ж дії, що і ***ADD приймач, 1***.

Команда ***DEC приймач*** виконує такі ж дії, що і ***SUB приймач, 1***.

Команда ***LEA r32, [адреса]*** записує в 32-розрядний реґістр значення адреси. Прапори не змінюються. Адреса записується в квадратних дужках – так можна визначити адреси змінних, що мають складну адресацію.

Наприклад, є глобальний масив ***y: array [0..1000] of longint***. Тоді після виконання команди ***LEA EAX, [y + 4 * 200]*** у реґістрі ***EAX*** буде зберігатися адреса комірки ***y [200]*** (тобто 200-й елемент масиву, починаючи з нуля). Множення на 4 потрібно, бо масив містить чотирибайтові дані. Також команду ***LEA*** можна використовувати для швидких обчислень виразів певного типу.

Команда ***CBW*** розширює ***AL*** до ***AX***. Команда ***CWD*** розширює ***AX*** до ***DX:AX***. Команда ***CWDE*** розширює ***AX*** до ***EAX***. Команда ***CDQ*** розширює ***EAX*** до ***EDX:EAX***. Якщо старший біт вихідного реґістру був одиницею, то відбувається розширення одиницями, інакше – розширення нулями. По суті ці команди розширюють число зі знаком.

Наприклад, є число 00111101b (тобто 61) типу shortint. Після виконання над ним команди ***CBW*** воно перетвориться в число 0000000000111101 (тобто теж 61). Якщо число було негативним, наприклад 10001000b (тобто -120), то після розширення воно буде 1111111110001000b (тобто теж 120). Таким чином, ці команди розширюють числа зі знаком (щоб розширити число без знака, треба заповнити нулями частину, що доповнюється; для цього існують спеціальні команди).

Команда ***MOVSX приймач, джерело*** копіює вміст джерела (r8 або m8/r16 або m16) у приймач (r16/r32) і розширює знак аналогічно ***CBW/CWDE***. Команда ***MOVZX*** копіює вміст джерела (r8 або m8/r16 або m16) в приймач (r16/r32) і розширює число нулями. По суті розширює число без знака.

Команда ***BSWAP r32*** міняє місцями перший і четвертий байти в 32-розрядному реґістрі, а також другий і третій. Наприклад, якщо реґістр ***EAX*** = 12345678h, то після виконання команди ***BSWAP EAX*** реґістр ***EAX*** = 78563412.

Команда ***IN приймач, джерело*** копіює в приймач (***AL/AX/EAX***) число з порту введення-виведення, номер якого вказаний у джерелі. Джерело – реґістр ***DX***, або восьмибітна константа.

Команда ***OUT джерело, приймач*** копіює число з порту з номером, указаним у джерелі (реґістр ***DX*** або восьмибітна константа), у приймач (***AL/AX/EAX***).

Команди ***IN*** і ***OUT*** зазвичай використовуються драйверами різних пристроїв (рулі, джойстики, принтери, сканери, тощо). Також вони використовуються обробниками переривань. Коли ми пишемо ***int 21h***, то відбувається насправді безліч дій (у тому числі – безліч виконань ***IN*** і ***OUT***).

Команда ***IMUL джерело*** виконує знакове множення. Джерело – r8/r16/r32/m8/m16/m32. Джерело множиться на ***AL/AX/EAX*** залежно від розрядності операнда і результат розташовується в ***AX/DX: AX/EDX: EAX*** відповідно. Якщо результат розташувався у молодшій половині, то ***CF*** = = ***OF*** = 0, інакше – ***CF*** = ***OF*** = 1. Прапори ***ZF***, ***SF***, ***AF***, ***PF*** не визначені.

Команда множення ***MUL/IMUL*** передбачає, що результат в два рази довше множників.

Команда ***MUL джерело*** виконує беззнакове множення (аналогічно попередньому).

Команда ***IDIV джерело*** – цілочислове ділення зі знаком. Джерело – r8/r16/r32/m8/m16/m32. Цілочислове ділення зі знаком виконується таким чином:

8 біт: ***AX***/ x8; ***AL*** – частка; ***AH*** – залишок

16 біт: ***DX***: ax/ x16; ***AX*** – частка; ***DX*** – залишок

32 біт: ***EDX***: ***EAX***/ x32; ***EAX*** – частка; ***EDX*** – залишок

Прапори ***ZF***, ***SF***, ***CF***, ***OF***, ***AF***, ***PF*** після цієї команди не визначені. При переповненні або діленні на нуль програма аварійно завершується. Наприклад, якщо ***AX*** = 256, ***CL*** = 1, то після команди ***DIV CL*** в ***AL*** має записатися число 256, а це неможливо (реґістр восьмибітний). Тому програма завершиться аварійно.

Команда ***DIV джерело*** – беззнакове ділення (аналогічна команді ***IDIV***).

Команда ***IMUL приймач, джерело*** виконує множення джерела (C/r/m) на приймач (реґістр), результат заносить у приймач. Розрядність операндів – 16 або 32.

Команда ***IMUL приймач, джерело1, джерело2*** – джерело1 (r/m) множиться на джерело2 (число), результат заноситься в приймач (реґістр). Розрядність операндів – 16 або 32. Якщо сталося переповнення і втрата старших бітів результату, то ***OF*** = ***CF*** = 1, інакше – ***OF*** = ***CF*** = 0. Значення ***SF***, ***ZF***, ***AF***, ***PF*** не визначені.

Команда ***NEG приймач*** виконує над числом, що міститься в приймачі (r8/r16/32/m8/m16/m32), операцію доповнення до двох. Ця операція еквівалентна оберненню знака, якщо розглядати приймач як число зі знаком. Якщо приймач дорівнює нулю, то ***CF*** = 0, інакше – ***CF*** = 1. Решта прапорів (***OF***, ***SF***, ***ZF***, ***AF***, ***PF***) призначаються відповідно до результату операції. Якщо не зважати на прапори, то команда ***NEG*** по суті рівносильна послідовному виконанню команди ***NOT***, потім – ***INC***. Приклад застосування команди ***NEG*** – отримання абсолютного значення числа @label: neg eax; js @label.

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання. Обчислити вираз *(c+4*d-123)/(1-a/2)*. Розмір змінної *a* – один байт, розмір *c* і *d* – два байти.

Лістинг програми такий:

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <fstream>

using namespace std;

void main()
{
	//creating the necessary variables
	short int u, c, d;//declaration variables "u", "c", "d"
	short int res, res1;//declaration variables "res" and "res1", to store the result
	//read variable values from file
	fstream file_in;//creating object for input from file
	file_in.open("filein.txt");//open file "filein.txt" for reading values "a" and "mask"
	file_in >> u >> c >> d;//read variables "u", "c", "d"
	file_in.close();//close the file after reading the variables
	char a = char(u);//convetting variable "u" (type is short int) to variable "a" (type is
char) because the variable "a" must be one-byte-type
	//using Assembler language
	_asm
	{                           //the beginning of the assembler insert
		mov al, a  //put in the register al the value of the variable "a"
		sar al, 1  //1 bit right or the value of register al is "a/2"
		cbw        //expand register al to register ax
		mov bx, 1  //put in the register al the value "1"
		sub bx, ax //the value of register bx is the difference of the value of resister
bx and the value of resister ax
		mov cx, bx //put in the register cx the value of the register bx
		//denominator

		mov ax, c   //put in the register ax the value of the variable "c"
		mov bx, d   //put in the register bx the value of the variable "d"
		sal bx, 2   //2 bits left or the value of register bx is "4*d"
		add ax, bx  //the value of register ax is the addition of the value of resister
ax and the value of resister bx
		sub ax, 123 //the value of register ax is the difference of the value of resister
ax and the value "123"
		cwd        //expand register ax to the couple of registers ax:dx
		//numerator

		idiv cx    //the value of the couple of registers ax:dx diveded by the value of
register cx
		//the quotient of division is in the register ax, the remainder of division is in
the register dx
		mov res, ax    //assign the variable "res" result of a logical operation
		mov res1, dx   //assign the variable "res1" result of a logical operation
	}                           //end of assembler insert
	//output to the file
	fstream file_out;   // creating object for write results to file
	file_out.open("fileout.txt");   //opening file "fileout.txt" to record results
	file_out << "quotient: " << res << endl;
	file_out << "remainder: " << res1 << endl;  //record to file value the variables "res"
and "res1"
	file_out.close();   //close the file after record the variables
	//output to console
	cout << "quotient: " << res << endl;
	cout << "remainder: " << res1 << endl;  //record to console value the variables "res"
and "res1"
	system("pause");//waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 7.1, а результат виконання програми – на рисунках 7.2 і 7.3.

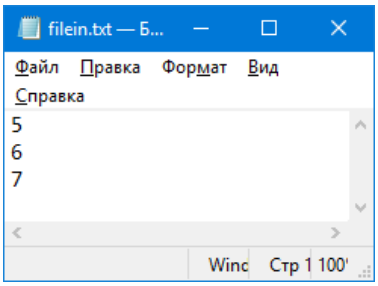


Рисунок 7.1 – Файл із початковими номерами та масками

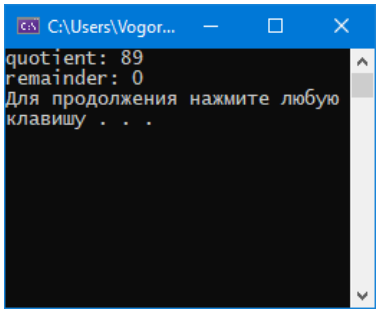


Рисунок 7.2 – Результат виконання у консолі

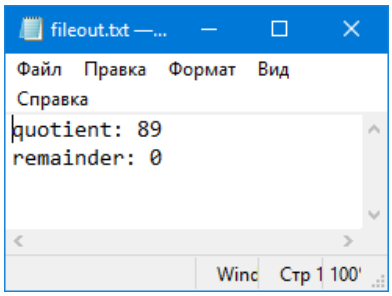


Рисунок 7.3 – Результат виконання у файлі

Варіанти завдань

Обчислити змішаний арифметичний вираз (чисельник/знаменник), використовуючи задані розміри операндів (відповідні типи даних у байтах). Варіанти завдань наведено у таблиці 7.1.

Таблиця 7.1 – Варіанти завдань

Номер варіанта	Чисельник	Знаменник	Розмірність операндів у байтах			
			a	b	c	d
1	2*c + b*23 - 12	a/4 - d	1	1	2	2
2	-8*c - d*15 + 15	a + b/8	1	1	2	2
3	c*5 - d/4 + 83	2*a - b	1	1	2	2
4	11*c - d*7 + 15	2*a - b	1	1	2	2
5	2*a + d*23	c + b - 1	1	1	2	2

Продовження таблиці 7.1

Номер варіанта	Чисельник	Знаменник	Розмірність операндів у байтах			
			a	b	c	d
6	a + d/4 + 253	c + b*2 - 2	1	1	2	2
7	a + d*5	c/4 + b - 3	1	1	2	2
8	c*(b + 23)	a/4 + d - 1	1	1	2	2
9	a*d + 23	c + b/4 - 3	1	1	2	2
10	b*c + d	a*23	1	1	2	2
11	2*c + b*23 - 12	a/4 - d	1	2	1	2
12	-8*c - d*15 + 15	a + b/8	1	2	1	2
13	c*5 - d/4 + 83	2*a - b	1	2	1	2
14	11*c - d*7 + 15	2*a - b	1	2	1	2
15	2*a + d*23	c + b - 1	1	2	1	2
16	a + d/4 + 253	c + b*2 - 2	1	2	1	2
17	a + d*5	c/4 + b - 3	1	2	1	2
18	c*(b + 23)	a/4 + d - 1	1	2	1	2
19	a*d + 23	c + b/4 - 3	1	2	1	2
20	b*c + d	a*23	1	2	1	2
21	2*c + b*23 - 12	a/4 - d	2	1	2	1
22	-8*c - d*15 + 15	a + b/8	2	1	2	1
23	c*5 - d/4 + 83	2*a - b	2	1	2	1
24	11*c - d*7 + 15	2*a - b	2	1	2	1
25	2*a + d*23	c + b - 1	2	1	2	1
26	a + d/4 + 253	c + b*2 - 2	2	1	2	1
27	a + d*5	c/4 + b - 3	2	1	2	1
28	c*(b + 23)	a/4 + d - 1	2	1	2	1
29	a*d + 23	c + b/4 - 3	2	1	2	1
30	b*c + d	a*23	2	1	2	1
31	2*c + b*23 - 12	a/4 - d	1	2	2	1
32	-8*c - d*15 + 15	a + b/8	1	2	2	1
33	c*5 - d/4 + 83	2*a - b	1	2	2	1
34	11*c - d*7 + 15	2*a - b	1	2	2	1
35	2*a + d*23	c + b - 1	1	2	2	1
36	a + d/4 + 253	c + b*2 - 2	1	2	2	1
37	a + d*5	c/4 + b - 3	1	2	2	1
38	c*(b + 23)	a/4 + d - 1	1	2	2	1
39	a*d + 23	c + b/4 - 3	1	2	2	1
40	b*c + d	a*23	1	2	2	1
41	2*c + b*23 - 12	a/4 - d	2	2	1	1
42	-8*c - d*15 + 15	a + b/8	2	2	1	1
43	c*5 - d/4 + 83	2*a - b	2	2	1	1
44	11*c - d*7 + 15	2*a - b	2	2	1	1

Продовження таблиці 7.1

Номер варіанта	Чисельник	Знаменник	Розмірність операндів у байтах			
			a	b	c	d
45	2*a + d*23	c + b - 1	2	2	1	1
46	a + d/4 + 253	c + b*2 - 2	2	2	1	1
47	a + d*5	c/4 + b - 3	2	2	1	1
48	c*(b + 23)	a/4 + d - 1	2	2	1	1
49	a*d + 23	c + b/4 - 3	2	2	1	1
50	b*c + d	a*23	2	2	1	1

Лабораторна робота № 8

ОРГАНІЗАЦІЯ УМОВНИХ І БЕЗУМОВНИХ ПЕРЕХОДІВ

Мета роботи: ознайомитися з принципами організації безумовних і умовних переходів у мові Assembler; вивчити команди умовного переходу, що виконуються після команди порівняння, команди умовного переходу, що виконуються залежно від значення будь-якого прапора, команди умовного переходу залежно від значення реґістра ***ECX***, ***CX***.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.
2. Проаналізувати завдання відповідно до варіанта. Визначити значення змінних, для яких можуть бути отримані правильні результати. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.
3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран у десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.
4. Протестувати програму на різних вхідних даних.
5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістінг програмного коду.

Теоретичні відомості

Команда безумовного переходу має такий синтаксис:

JMP <операнд>.

Операнд вказує адресу переходу. Існує два способи указання цієї адреси, відповідно розрізняють прямий і непрямий переходи.

Якщо в команді переходу вказується мітка команди, до якої потрібно перейти, то перехід називається *прямим*:

JMP L

...

...

L: MOV EAX, x.

При *непрямому* переході в команді переходу вказується не адреса переходу, а реґістр або елемент пам'яті, де ця адреса знаходиться. Вміст зазначеного реґістра або комірки пам'яті розглядається як абсолютна адреса переходу. Непрямі переходи використовуються в тих випадках, коли адреса переходу стає відомою тільки під час роботи програми:

JMP EBX.

Розглянемо умовний перехід. У системі команд процесора архітектури x86 не передбачена підтримка умовних логічних структур, характерних для мов високого рівня. Однак мовою Assembler за допомогою набору команд порівняння та умовного переходу можна реалізувати логічну структуру будь-якої складності. У мові високого рівня будь-який умовний оператор виконується в два етапи. Спочатку обчислюється значення умовного виразу, а потім залежно від його результату виконуються ті чи інші дії. Проводячи аналогію з мовою Assembler, можна сказати, що спочатку виконуються такі команди, як ***CMP***, ***AND*** або ***SUB***, що впливають на прапори стану процесора. Потім виконується команда умовного переходу, яка аналізує значення потрібних прапорів, і в разі якщо вони встановлені, виконується перехід за вказаною адресою.

Команд умовного переходу досить багато, але всі вони записуються за зразком:

JCC <мітка>.

Тут ***J*** – jump, а ***CC*** – мнемокод команди.

Усі команди умовного переходу можна розділити на три групи.

До першої групи належать команди, які зазвичай використовуються після виконання команди порівняння. В їх мнемокодах вказується той результат порівняння, при якому треба виконати перехід:

E – Equal (еквівалентність);

N – Not (заперечення);

G – Greater (більше; застосовується для чисел зі знаком);

L – Less (менше; застосовується для чисел зі знаком);

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

B – Bellow (нижче, менше; застосовується для чисел без знака).

У таблиці 8.1 наведено приклади першої групи команд умовного переходу.

Таблиця 8.1 – Команди умовного переходу першої групи

Мнемокод	Значення мнемокоду	Умова переходу після команди **CMP op1, op2**	Значення прапорів
Для всіх чисел			
JE	Перехід, якщо дорівнює	op1 = op2	ZF = 1
JNE	Перехід, якщо не дорівнює	op1 ≠ op2	ZF = 0
Для чисел зі знаком			
JL/JNGE	Перехід, якщо менше	op1 < op2	SF ≠ OF
JLE/JNG	Перехід, якщо менше або дорівнює	op1 ≤ op2	SF ≠ OF або ZF = 1
JG/JNLE	Перехід, якщо більше	op1 > op2	SF = OF і ZF = 0
JGE/JNL	Перехід, якщо більше або дорівнює	op1 ≥ op2	SF = OF
Для чисел без знака			
JB/JNAE	Перехід, якщо нижче	op1 < op2	CF = 1
JBE/JNA	Перехід, якщо нижче або дорівнює	op1 ≤ op2	CF = 1 або ZF = 1
JA/JNBE	Перехід, якщо вище	op1 > op2	CF = 0 і ZF = 0
JAE/JNB	Перехід, якщо вище або дорівнює	op1 ≥ op2	CF = 0

До другої групи команд умовного переходу належать команди, які зазвичай ставляться після команд, відмінних від команди порівняння, і які реагують на те чи інше значення будь-якого прапора (таблиця 8.2).

Таблиця 8.2 – Команди умовного переходу другої групи

Мнемокод	Умова переходу	Мнемокод	Умова переходу
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

До третьої групи належать дві команди умовного переходу, що перевіряють не прапори, а значення реґістра ***ECX*** або ***CX***:

JCXZ <мітка>; – перехід, якщо значення реґістра ***CX*** дорівнює 0.
JECXZ <мітка>; – перехід, якщо значення реґістра ***ECX*** дорівнює 0.

Однак ці команди виконуються досить довго. Вигідніше провести порівняння з нулем і використати звичайну команду умовного переходу.

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання. Обчислити значення арифметичного виразу *res* залежно від виконання умови

$$X = \begin{cases} \frac{a}{b} + 1, \text{коли } a < b; \\ -1, \text{коли } a = b \ ; \\ \frac{a * b - 5}{a}, \text{коли } a > b. \end{cases}$$

Лістинг програми такий:

```
#include <iostream>
#include <windows.h>
#include <cstdlib>
#include <conio.h>
#include <fstream>

using namespace std;

void main()
{
	//creating the necessary variables
	long int x = 0, x1 = 0;	//declaration variable "x" and "x1", to store the result
	int a, b;			//declaration variable "a" and "b"
							//read variable values from file
	ifstream file_in;		//creating object for input from file
	file_in.open("filein.txt");//open file "file_in.txt" for reading values "a" and "b"
	file_in >> a;			//read variable "a"
	file_in >> b;			//read variable "b"
	file_in.close();		//close file "file_in.txt" after reading the variables

	_asm			//using Assembler language
	{
		mov eax, a	//put in the register eax the value of the variable "a"
		cmp eax, b	//compare  variable "a" with variable "b"
		je _equal	//would jump to label "_equal" if and only if "a" and "b" were equal
		jg _greater //would jump to label "_greater" if and only if "a" was greater than
"b"
		jl _lower	//would jump to label "_lower" if and only if "a" was lower than "b"

		_equal :	//label, which would go off if and only if "a" and "b" were equal
		mov eax, -1	//put in the register eax number -1
			mov x, eax	//put in the variable "x" the value of the register eax
(assign the variable "x" result of this label)
			jmp _exitt	//would jump to label "_exitt"

			_greater :	//label, which would go off if and only if "a" was greater
than "b"
```

```
            mov ebx, b      //put in the register ebx number the value of the variable "b"
                  imul ebx        //multiply "eax" by "ebx" -> eax = a*b
                  sub eax, 5      //substract 5 from eax (actually we substract 5 from
variable which is stored in register eax)
                  cdq             //Convert Doubleword to Quadword -> eax to eax:edx
                                          //quotient is stored in the register eax,
remainder is stored in the register edx
                  idiv a          //performs signed integer division -> eax:edx = eax:edx/ a
                  mov x, eax      //put in the variable "x" the value of the register eax
(assign the variable "x" result of this label (quotient))

                  mov x1, edx     //put in the variable "x1" the value of the register edx
(assign the variable "x1" result of this label (remainder of devision))
                  jmp _exitt      //would jump to label "_exitt"


                  _lower :      //label, which would go off if and only if "a" was lower than
"b"
            cdq          //Convert Doubleword to Quadword -> eax to eax:edx
                                  //quotient is stored in the register eax, remainder is stored
in the register edx
                  idiv b       //performs signed integer division -> eax:edx = eax:edx/ b
                  inc eax      //eax = eax + 1
                  mov x, eax   //put in the variable "x" the value of the register eax
(assign the variable "x" result of this label (quotient))
                  jmp _exitt   //would jump to label "_exitt"

                  _exitt : mov ax, 4C00h;// DOS exit function

      }//end of assembler insert
      //output to the file
      ofstream file_out;                        // creating object for write results to file
      file_out.open("fileout.txt");             //opening file "fileout.txt" to record results
      file_out << "x ="  << x << endl;          //record to file value the variable "x"
      file_out << "x1 =" << x1;                 //record to file value the variable "x1"
      file_out.close();                         //close file "fileout.txt"
      //output to console
      cout << "x =" << x << endl;               //record to file value the variable "x"
      cout << "x1 =" << x1;                     //record to file value the variable "x1"
      system("pause");//waiting to press the button "enter", for the end of the program
}
```

На рисунках 8.1 і 8.2 показано вхідний і вихідний файли для умови a>b, на рисунках 8.3 і 8.4 – вхідний і вихідний файли для умови a=b, на рисунках 8.5 і 8.6 – вхідний і вихідний файли для умови a<b.

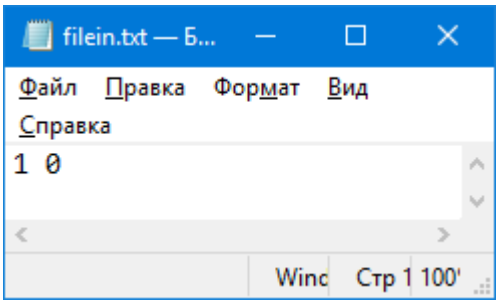


Рисунок 8.1 – Файл filein.txt

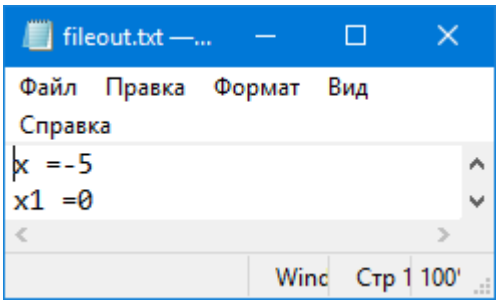


Рисунок 8.2 – Файл fileout.txt

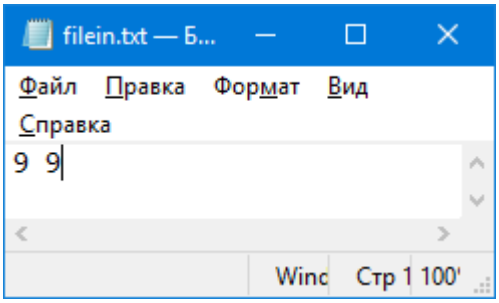


Рисунок 8.3 – Файл filein.txt

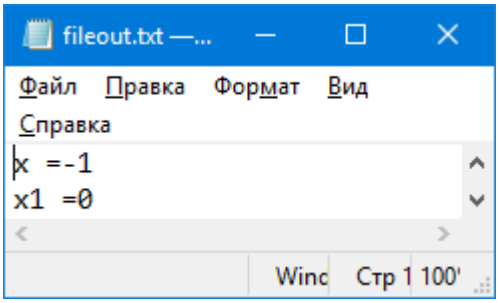


Рисунок 8.4 – Файл fileout.txt

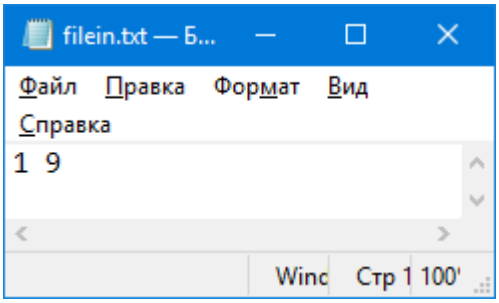


Рисунок 8.5 – Файл filein.txt

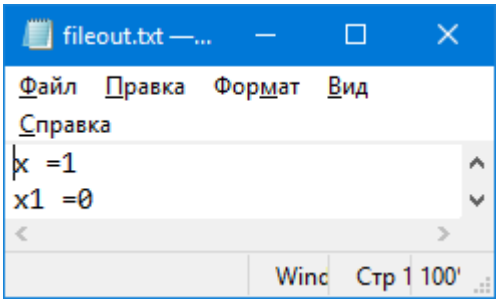


Рисунок 8.6 – Файл fileout.txt

Варіанти завдань

Використовуючи команди умовного переходу, обчислити значення арифметичного виразу *res* залежно від виконання певної умови. Варіанти завдань наведено у таблиці 8.3.

Таблиця 8.3 – Варіанти завдань

Номер варіанта	Умова			Розмірність операндів у байтах	
	Якщо a > b, то res =	Якщо a = b, то res =	Якщо a < b, то res =	a	b
1	a*b + 1	25	(a - 5)/b	1	1
2	a*b - 3	2	$(a^3 + 1)/b$	1	1
3	(a*a - b)/b	-5	a/b + 5	1	1
4	(a*b - 5)/a	-1	a/b + 1	1	1
5	a/b - 1	-25	$(a^3 - 5)/a$	1	1
6	a*b + 21	-5	3*a/b + 1	1	1
7	5*a + b	-125	(a - 5)/b	1	1
8	a*b - 1	255	(a - 5)/b	1	1
9	b*a + 1	-10	(a - 5)/b	1	1
10	a/b + 31	-25	(a*5 - 1)/a	1	1

Продовження таблиці 8.3

Номер варіанта	Умова			Розмірність операндів у байтах	
	Якщо a > b, то res =	Якщо a = b, то res =	Якщо a < b, то res =	a	b
11	2*a + b	-2	(a - 5)/b	1	1
12	$(a^3 - 5)/b$	25	b/a + 1	1	1
13	b/a + 61	-5	(b - a)/b	1	1
14	a/b + 1	-2	(a - b)/a	1	1
15	$(a^3 + b)/a$	-4	(a*3 - 5)/b	1	1
16	(a - 235)/b	-295	b/a - 1	1	1
17	2*a/b + 1	-445	(b + 5)/a	1	1
18	a/b + 1	a+25	(a*b - 2)/a	1	1
19	b*a + 1	3425	(2*a - 5)/b	1	1
20	a*a - b	-a	(a*b - 1)/b	1	1
21	a*b + 1	-b	(a - 5)/b	1	1
22	(b - 5)/a	25-a	a/b - 1	1	1
23	b/a + 2	-11	(a - 8)/b	1	1
24	a/b + 2	8	(b - 9)/a	1	1
25	(b - a)/b	-5	a*b + 5	1	1
26	(a - b)/a	-71	a/b + 1	1	1
27	b/a - 7	43	$(a^3 - b)/b$	1	1
28	(-5) + a/b	45	(3*a - 6)/b	1	1
29	a/b + 7	-125	(3*b + 9)/a	1	1
30	(b - 5)/a	-55	(a*b + 1)/4	1	1
31	a/b + 20	110	(a - b)/a	1	1
32	a/b + 11	-11	(3*b - 9)/a	1	1
33	b/a - 5	22	(a - 9)/b	1	1
34	2*a/b + 1	-5	$(a^3 - 9)/a$	1	1
35	b/a + 2	-57	(a - b)/b	1	1
36	b/a + 1	-20	(a - 45)/b	1	1
37	b/a + 1	-44	$(a^3 + b)/b$	1	1
38	a/b + 2	-100	(b - 100)/a	1	1
39	a/b + 201	-800	(a - 5)/a	1	1
40	a/b + 1	-100	(a*b - 9)/3	1	1
41	a/b + 1	-425	(3*a - b)/a	1	1
42	(a*b - 8)/a	-b	b*b - a	1	1
43	(b - 9)/a	-b	a*b + 5	1	1
44	a/b + 1	2*a	(a - 10)/b	1	1

Продовження таблиці 8.3

Номер варіанта	Умова			Розмірність операндів у байтах	
	Якщо a > b, то res =	Якщо a = b, то res =	Якщо a < b, то res =	a	b
45	a*b + 8	455	(a - b)/b	1	1
46	b/a + 1	-271	(a - b)/b	1	1
47	a/b - 37	3	$(a^3 - b)/a$	1	1
48	a*b - 5	-195	(a - 6)/b	1	1
49	a/b - 7	-195	$(a^3 + 9)/a$	1	1
50	$(b^3-5)/a$	-155	a + b/4	1	1

Лабораторна робота № 9

ОБРОБЛЕННЯ РЯДКІВ

Мета роботи: ознайомитися з поданням рядків у пам'яті обчислювальних машин і принципами їх оброблення мовою Assembler; вивчити організацію виконання ланцюгових команд.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.
2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.
3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран у десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.
4. Протестувати програму на різних вхідних даних.
5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістинг програмного коду.

Теоретичні відомості

Розглянемо операції з рядками.

Команда ***MOVSB/MOVSW/MOVSD*** копіює байт/слово/подвійне_слово з пам'яті за адресою ***ESI*** у пам'ять за адресою ***EDI***. Після виконання команди реґістри ***ESI*** і ***EDI*** збільшуються на 1/2/4, коли прапор ***DF*** = 0, і зменшуються, коли ***DF*** = 1. Команда ***MOVS*** з префіксом ***REP*** виконує копіювання рядка байтів/слів/подвійних_слів довжиною ***ECX***.

Команда ***CMPSB/CMPSW/CMPSD*** порівнює один байт/слово/подвійне_слово з пам'яті за адресою ***ESI*** з байтом/ словом/подвійним_словом за адресою ***EDI*** і встановлює прапори аналогічно команді ***CMP***. При цьому ***ESI*** і ***EDI*** просуваються. Команда ***CMPSB/MPSW/CMPSD*** з префіксами ***REPNE/REPNZ*** або ***REPE/REPZ*** виконує порівняння рядка довжиною в ***ECX*** байтів/слів/подвійних_слів. У першому випадку порівняння триває до першого збігу в рядках, у другому – до першої розбіжності. Команда ***SCASB/SCASW/SCASD*** порівнює ***AL/AX/EAX*** з байтом/словом/подвійним_словом з пам'яті за адресою ***EDI*** і встановлює прапори аналогічно команді ***CMP***. При цьому ***EDI*** просувається.

Команда ***LODSB/LODSW/LODSD*** копіює байт/слово/подвійне_слово з пам'яті за адресою ***ESI*** в ***AL/AX/EAX***. При цьому ***ESI*** просувається. Команда ***LODSB/LODSW/LODSD*** з префіксом ***REP*** виконує копіювання рядка довжиною в ***ECX***, і в акумуляторі (реґістрі ***AL/AX/EAX***) опиниться останній елемент рядка. Насправді ***LODS*** використовують без префіксів, часто всередині циклу в парі з командою ***STOS***.

Команда ***STOSB/STOSW/STOSD*** копіює ***AL/AX/EAX*** у пам'ять за адресою ***EDI***. При цьому ***EDI*** просувається.

Команда ***INSB/INSW/INSD*** виконує зчитування рядка байтів/ слів/подвійних_слів з порту введення-виведення, номер якого вказаний у ***DX***, у пам'ять за адресою ***EDI***. При цьому ***EDI*** просувається. Цю команду можна замінити за допомогою використання команди ***IN*** і організації циклів.

Команда ***OUTSB/OUTSW/OUTSD*** записує у порт введення-виведення, номер якого вказано в ***DX***, байт/слово/подвійне_слово з пам'яті за адресою ***ESI***. Починаючи з процесорів pentium перевіряється готовність порту прийняти нові дані при виконанні команди з префіксом ***REP***, тому дані не втрачаються. При цьому ***ESI*** просувається. Команду можна замінити за допомогою використання команди ***OUT*** і організації циклів.

Перед командами оброблення рядків можна ставити префікс ***REP***. Тоді команда буде повторюватися стільки разів, скільки вказано в ***ECX*** (може бути нуль разів). Якщо вказати префікс ***REPE/REPZ***, буде те ж саме (***ECX*** повториться), але при ***ZF*** = 0 повторення будуть припинені. Перевірка ***ZF*** відбувається після виконання команди. Наприклад, якщо ***ZF*** = 0, при

цьому ***ECX*** <> 0, то ***REPE MOVSB*** виконається один раз. Команди ***REPNE/REPNZ*** перестануть повторюватися, якщо ***ZF*** = 1. Префікс ***REP*** зазвичай використовується з ***MOVS***, ***STOS***, ***LODS***, ***INS***, ***OUTS***, а команди ***REPE/REPZ***/***REPNE/REPNZ*** – з ***CMPS*** і ***SCAS***. Поведінку префіксів в інших випадках не визначено.

Розглянемо ланцюгові команди.

I. Пересилання ланцюжка (***MOV****e* ***S****tring*):

MOVS адреса_приймача, адреса_джерела.

MOVSB – переслати ланцюжок байтів; ***MOVSW*** – переслати ланцюжок слів; ***MOVSD*** – переслати ланцюжок подвійних слів. Команда копіює байт, слово або подвійне слово з ланцюжка, що адресується ***DS: SI***, у ланцюжок, що адресується ***DS: SI***. Для ***MOVS*** адреси приймача і джерела вказуються при виклику.

Алгоритм роботи команди такий.

1. Виконати копіювання байта, слова або подвійного слова з операнда джерела в операнд приймач, при цьому адреси елементів повинні бути попередньо завантажені:

– адреса джерела – в пару реґістрів ***DS: ESI/SI*** (***DS*** за замовчуванням, допускається заміна сегмента);

– адреса приймача – в пару реґістрів ***ES: EDI/DI*** (заміна сегмента не допускається).

2. Залежно від стану прапора ***DF*** змінити значення реґістрів ***ESI/SI*** і ***EDI/DI***.

3. Якщо вказаний префікс повторення, то виконати обумовлені ним дії.

4. Стан прапорів після виконання команди: команда не впливає на прапори.

II. Порівняння ланцюжків (***C****o****MP****are* ***S****tring*):

CMPS адреса_приймача, адреса_джерела.

CMPSB – порівняти рядок байтів; ***CMPSW*** – порівняти рядок слів; ***CMPSD*** – порівняти рядок подвійних слів. Команди, що реалізують операцію-примітив порівняння ланцюжків, виконують порівняння елементів, що знаходяться за адресою ***DS: SI***, з елементами за ***ES: DI***. Для ***CMPS*** адреси приймача і джерела вказуються при виклику.

Алгоритм роботи команди такий.

1. Виконати віднімання елементів (джерело – приймач), адреси елементів попередньо повинні бути завантажені:

– адреса джерела – в пару реґістрів ***DS: ESI/SI***;

– адреса призначення – в пару реґістрів ***ES: EDI/DI***.

2. Залежно від стану прапора ***DF*** змінити значення реґістрів ***ESI/SI*** і ***EDI/DI***.

3. Залежно від результату віднімання встановити прапори:

– якщо чергові елементи ланцюжків не дорівнюють один одному, то ***CF*** = 1, ***ZF*** = 0;

– якщо чергові елементи ланцюжків або ланцюжка в цілому рівні, то ***CF*** = 0, ***ZF*** = 1.

4. При наявності префікса виконати зумовлені ним дії.

III. Сканування ланцюжка (***SCA****ning* ***S****tring*):

SCAS адреса_приймача.

SCASB – сканувати ланцюжок байтів; ***SCASW*** – сканувати ланцюжок слів; ***SCASD*** – сканувати ланцюжок подвійних слів. Команди, що реалізують операцію-примітив сканування ланцюжків, проводять пошук значення, що знаходиться в реґістрі ***AL, AX*** або ***EAX*** (залежно від розміру) за адресою ***ES: DI***. Для ***SCAS*** адреса рядка, що сканується, вказується при виклику.

Алгоритм роботи команди такий.

1. Виконати віднімання (елемент ланцюжка – ***EAX/AX/AL***). Елемент ланцюжка локалізується парою ***ES: EDI/DI***. Заміна сегмента ***ES*** не допускається.

2. За результатом віднімання встановити прапори.

3. Змінити значення реґістра ***EDI/DI*** на величину, що дорівнює довжині елемента ланцюжка. Знак цієї величини залежить від стану прапора ***DF***.

IV. Завантаження елемента з ланцюжка (***LO****a****D*** ***S****tring*):

LODS адреса_джерела.

LODS – завантажити елемент з ланцюжка у реґістр-акумулятор ***AL/ AX/EAX***; ***LODSB*** – завантажити байт з ланцюжка у реґістр ***AL***; ***LODSW*** – завантажити слово з ланцюжка у реґістр ***AX***; ***LODSD*** – завантажити подвійне слово з ланцюжка у реґістр ***EAX***. Операція-примітив завантаження елемента ланцюжка в акумулятор дозволяє вилучити елемент за адресою ***DS: SI*** (для ***LODS*** адреса рядка джерела вказується при виклику) і помістити його в реґістр-акумулятор ***AL***, ***AX*** або ***EAX***.

Алгоритм роботи команди такий.

1. Завантажити елемент з елементу пам'яті, що адресується парою ***DS: ESI/SI***, у реґістр ***AL/AX/EAX***. Розмір елемента визначається неявно (для команди ***LODS***) або явно відповідно до застосованої команди (для команд ***LODSB***, ***LODSW***, ***LODSD***).

2. Змінити значення реґістра ***SI*** на величину, що дорівнює довжині елемента ланцюжка. Знак цієї величини залежить від стану прапора ***DF***.

3. Стан прапорів після виконання команди: команда не впливає на прапори.

V. Збереження елемента в ланцюжку (***STO****re* ***S****tring*):

STOS адреса_приймача.

STOS – зберегти в ланцюжку елемент з реґістра-акумулятора ***AL/AX/EAX***; ***STOSB*** – зберегти в ланцюжку байт з реґістра ***AL***; ***STOSW*** – зберегти в ланцюжку слово з реґістра ***AX***; ***STOSD*** – зберегти в ланцюжку подвійне слово з реґістра ***EAX***. Операція-примітив перенесення елемента з акумулятора в ланцюжок дозволяє зберегти значення з реґістра-акумулятора в елементі за адресою ***ES: DI***. Для ***STOS*** адреса рядка-приймача вказується при виклику.

Алгоритм роботи команди такий.

1. Записати елемент з реґістра ***AL/AX/EAX*** в елемент пам'яті, що адресується парою ***ES: DI/EDI***. Розмір елемента визначається неявно (для команди ***STOS***) або конкретною застосованою командою (для команд ***STOSB***, ***STOSW***, ***STOSD***).

2. Змінити значення реґістра ***DI*** на величину, що дорівнює довжині елемента ланцюжка. Знак цієї зміни залежить від стану прапора ***DF***.

3. Стан прапорів після виконання команди: команда не впливає на прапори.

VI. Отримання елементів ланцюжка з порту введення-виведення (***IN****put* ***S****tring*):

INS адреса_приймача, номер_порту.

INSB – увести з порту ланцюжок байтів; ***INSW*** – увести з порту ланцюжок слів; ***INSD*** – увести з порту ланцюжок подвійних слів. Ця команда вводить елемент з порту, номер якого знаходиться в реґістрі ***DX***, в елемент ланцюжка за адресою ***ES: DI***. Для ***INS*** адреса рядка-приймача і номер порту вказуються при виклику.

Алгоритм роботи команди такий.

1. Передати дані з порту введення-виведення, номер якого завантажений в реґістр ***DX***, у пам'ять за адресою ***ES: EDI/DI***.

2. Залежно від стану прапора ***DF*** змінити значення реґістрів ***EDI/DI***.

3. При наявності префікса виконати зумовлені ним дії.

4. Стан прапорів після виконання команди: команда не впливає на прапори.

VII. Виведення елементів ланцюжка в порт введення-виведення (***Out**put **S**tring*):

OUTS адреса_джерела, номер_порту.

OUTBS – вивести ланцюжок байтів у порт введення-виведення; ***OUTWS*** – вивести ланцюжок слів у порт введення-виведення; ***OUTDS*** – вивесті ланцюжок подвійних слів у порт введення-виведення. Ця команда виводить елемент за адресою ***DS: SI*** у порт, номер якого знаходиться в реґістрі ***DX***. Для ***OUTS*** адреса рядка-джерела і номер порту вказуються при виклику.

Алгоритм роботи команди такий.

1. Передати дані в порт введення-виведення, номер якого завантажений в реґістр ***DX***, з комірки пам'яті за адресою ***DS: ESI/SI***.

2. Залежно від стану прапора ***DF*** змінити значення реґістрів ***ESI/SI***.

3. При наявності префікса виконати зумовлені ним дії.

4. Стан прапорів після виконання команди: команда не впливає на прапори.

Набір команд процесора має відповідні машинні команди тільки для ланцюгових команд мови Assembler без операндів. Команди з операндами транслятор Assembler задіює тільки для визначення типів операндів. Після того як з'ясовано тип елементів ланцюжків за їх описом у пам'яті, генерується одна з трьох машинних команд для кожної з ланцюгових операцій. З цієї причини всі реґістри, що містять адреси ланцюжків, повинні бути ініційовані заздалегідь, у тому числі і для команд, що допускають явну вказівку операндів.

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання 1. Переслати 8 байтів з одного рядка в інший. Реґістри ***DS*** і ***ES*** ініціалізовані адресою сегмента даних.

Лістинг програми такий:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
	const int N = 20;				//the size of the strings str and str2;
	char str[N] = "";	// we use char string(C) str(size=20);
	char str2[N] = "";				// string(C) str2(size=20)  8 bytes from the
string str will be sent to the str2 string;
	ifstream file_in;				//creating object for input from file
	file_in.open("filein.txt");	//open file "file_in.txt" for reading values "str"
	file_in.getline(str, N - 1, '\n');//read variable "str"
	file_in.close();				//close file "file_in.txt" after reading the
variables
```

```
        __asm                                                   //assembler insertion;
        {
                mov ecx, 8;                                     //we put 8 in the register ecx,which has
a 32 bit size;
                                                                        //lea(Load Effective Address);
                lea edi, str2;                                  //used to obtain the effective source
address (offset);
                lea esi, str;                           //This command is an alternative to the
assembler offset operator.
                                                                        //In contrast to offset the lea
command allows indexing of operand that allows a more
                                                                        //flexible way to organize the
addressing of the operands.;
                jecxz label;                            //jecxz - (Jump if condition) Jump if ECX=Zero;
                                                                        //transition within the current
command segment depending on some condition

        loo:                                                    //the label of the transition;
                mov al, [esi];                                  //the esi register element is written to
the al register;
                mov[edi], al;                           //the al register is written to the esi
register element;
                                                                        //The brackets in the command MOV
means for
                                                                        //memory access uses the address
in the register specified in
                                                                        //square brackets.
                inc esi;                                        //inc(INCrement operand by 1) increase
the value of the operand in memory or register 1;
                inc edi;
                loop loo;                                       //LOOP control by register cx;
                                                                        //used to organize a loop with a
counter in the cx register.;
        label:                                                  //the label of the transition;
        }
        ofstream file_out;                              // creating object for write results to file
        file_out.open("fileout.txt");                   //opening file "fileout.txt" to record results
        file_out << "str  =" << str << endl;             //record to file value the variable "str"
        file_out << "str2 =" << str2 << endl;            //record to file value the variable "str2"
        file_out.close();                               //close file "fileout.txt"
        //output to console
        cout << "str  =" << str << endl;
        cout << "str2 =" << str2 << endl;

        system("pause");//waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 9.1, а результати виконання програми – на рисунках 9.2 і 9.3.

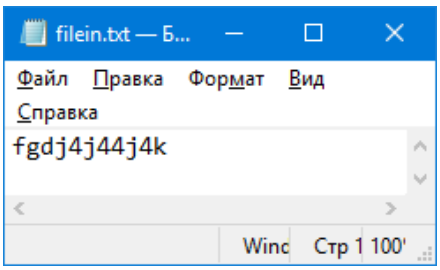


Рисунок 9.1 – Вхідний файл

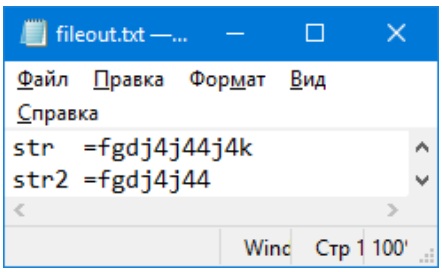


Рисунок 9.2 – Файл результату

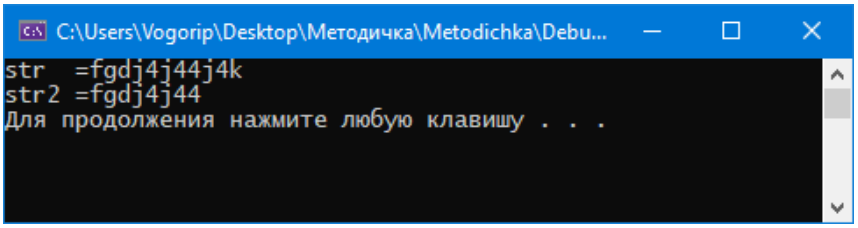


Рисунок 9.3 – Результат програми в консолі

Завдання 2. Порівняти два рядки однакової довжини (21 байт) побайтово зліва направо; операції припиняються, коли виявлено, що елементи рядків не дорівнюють один одному.

Лістинг програми такий.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
	const int N = 22;                                        //the size of the strings str and
str2;
	char str[N] = "";    //we use char string(C) str(size=20);
	char str2[N] = "";   //string(C) str2(size=20);

	ifstream file_in;                          //creating object for input from file
	file_in.open("filein.txt");       //open file "file_in.txt" for reading values "str"
	file_in >> str;
	file_in >> str2;
	file_in.close();                           //close file "file_in.txt" after reading the
variables
	short int n = 0;
	__asm                                                          //assembler insertion;
	{
		mov ecx, 21;                //we put 21 in the register ecx,which has a 32 bit
size;
                                                        //lea(Load Effective Address);
		lea esi, str;               //used to obtain the effective source address
(offset);

		lea edi, str2;                     //This command is an alternative to the
assembler offset operator.
                                                        //In contrast to offset the lea command
allows indexing of operand that allows a more
                                                        //flexible way to organize the
addressing of the operands.;
		repe cmpsb;                        //repe - (REPeat string operation);
                                                        //specify conditional and unconditional
repetition following this command;
		//Repe and repz actions:
		//the analysis of the content cx and zf flag :
		//if cx < >0 or zf < >0, execute the chain command following this prefix and
proceed to step 2;
		//if cx = 0 or zf = 0, then transfer control to the command following this chain
command(exit the loop by rep);
		//cmpsb - (CoMPare String Byte);
		//Purpose: comparison of two sequences(chains) of elements in memory.;
		jne label;          //Conditional jump commands are useful for checking various
conditions that occur during program execution.
```

```
            //As you know, many commands form the results of their work in the register
eflags/flags.
            //(jne)status of checked flags:ZF=0,transition condition:if not equal;
            //If the rows are equal, n will increase by 1,else the program will go to the
label "label";
            mov ax, n;
            inc ax;                    //inc(INCrement operand by 1) increase the value of
the operand in memory or register 1;
            mov n, ax;
            jmp mend;//Purpose: used in the program to organize unconditional transition both
within the current team segment and beyond.
      label:
      mend:
      }

      ofstream file_out;                        // creating object for write results to file
      file_out.open("fileout.txt");             //opening file "fileout.txt" to record results
      file_out << "str  =" << str << endl;       //record to file value the variable "str"
      file_out << "str2 =" << str2 << endl;      //record to file value the variable "str2"

      //output to console
      cout << "str: " << str << endl;
      cout << "str2:" << str2 << endl;
      if (n == 1)
      {
            file_out << "\nStrings are equal." << endl;
            cout << "\nStrings are equal." << endl;
      }
      else
      {
            file_out << "\nstrings are not equal." << endl;
            cout << "\nstrings are not equal." << endl;
      }

      file_out.close();                         //close file "fileout.txt"
      system("pause");//waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 9.4, а результати виконання програми – на рисунках 9.5 і 9.6.

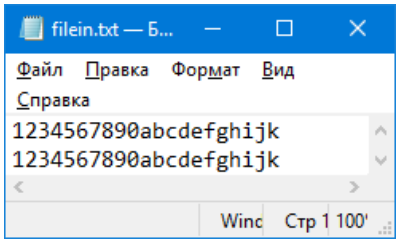


Рисунок 9.4 – Вхідний файл

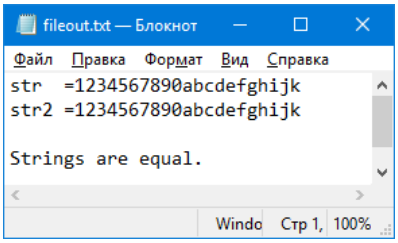


Рисунок 9.5 – Файл результату

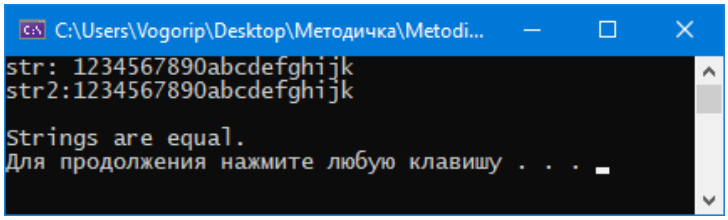


Рисунок 9.6 – Результат виконання програми в консолі

Варіанти завдань

Завдання 1. Знайти і замінити всі задані символи у вихідному рядку на інші символи. Розмір рядка – 50 символів. Варіанти завдань наведено у таблиці 9.1.

Завдання 2. Обчислити кількість елементів масиву, що задовольняють умову. Варіанти завдань наведено у таблиці 9.2.

Таблиця 9.1 – Варіанти завдання 1

Номер варіанта	Знайти символи	Замінити на символи	Розмір елемента рядка в байтах
1	10	0	1
2	15	1	2
3	20	2	4
4	7	3	1
5	11	4	2
6	17	5	4
7	23	6	1
8	4	7	2
9	6	8	4
10	9	9	1
11	0	0	2
12	0	1	4
13	0	2	1
14	=	3	2
15	/	4	4
16	|	5	1
17	\	6	2
18	.	7	4
19	,	8	1
20	<	9	2
21	>	0	4
22	"	1	1
23	:	2	2
24	;	3	4
25	№	4	1
26	!	5	2
27	@	6	4
28	#	7	1
29	$	8	2
30	%	9	4
31	^	0	1

Продовження таблиці 9.1

Номер варіанта	Знайти символи	Замінити на символи	Розмір елемента рядка в байтах
32	&	1	2
33	*	2	4
34	(	3	1
35	)	4	2
36	_	5	4
37	-	6	1
38	+	7	2
39	=	8	4
40	/	9	1
41	|	0	2
42	\	1	4
43	.	2	1
44	,	3	2
45	<	4	4
46	>	5	1
47	"	6	2
48	:	7	4
49	;	8	1
50	№	9	2

Таблиця 9.2 – Варіанти завдання 2

Номер варіанта	Знайти елементи	Кількість елементів у масиві	Розмір елементу масиву в байтах
1	= 10	5	4
2	> 10	10	1
3	< 10	15	2
4	= -5	20	4
5	> -5	7	1
6	< -5	13	2
7	= -3	17	4
8	>= -3	23	1
9	<= -3	4	2
10	= 7	6	4
11	>= 7	11	1
12	<= 7	16	2
13	= 15	21	4
14	> 15	8	1

Продовження таблиці 9.2

Номер варіанта	Знайти елементи	Кількість елементів у масиві	Розмір елементу масиву в байтах
15	<15	14	2
16	= -15	18	4
17	> -15	24	1
18	< -15	3	2
19	= -1	7	4
20	>= 10	13	1
21	<= 10	19	2
22	= 0	10	4
23	>= -7	15	1
24	<= -7	20	2
25	= -2	7	4
26	>= -9	13	1
27	<= -9	17	2
28	= 1	23	4
29	>= 9	4	1
30	<= 9	6	2
31	= 3	11	4
32	> 0	16	1
33	< 0	21	2
34	= -13	8	4
35	> -13	14	1
36	< -13	18	2
37	= -15	24	4
38	>= -5	3	1
39	<= -5	7	2
40	= 12	13	4
41	> 12	19	1
42	< 12	18	2
43	= -8	24	4
44	>= -5	3	1
45	<= -5	7	2
46	= -10	13	4
47	>= 10	19	1
48	<= 10	14	2
49	>= -10	18	4
50	<= -10	24	1

Лабораторна робота № 10

ОРГАНІЗАЦІЯ ЦИКЛІВ І РОБОТА З МАСИВАМИ ДАНИХ

Мета роботи: вивчити способи адресації і обчислення ефективної адреси в пам'яті обчислювальної машини; ознайомитися з поданням багатовимірних масивів у пам'яті і засвоїти принципи роботи циклів мовою Assembler.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.
2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.
3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран у десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.
4. Протестувати програму на різних вхідних даних.
5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістинг програмного коду.

Теоретичні відомості

Масив – упорядкована послідовність елементів одного типу. Для роботи з масивами необхідно вміти визначати масив і форматувати його елементи, здійснювати доступ до елементів. Для цього використовують різні способи адресації.

Розглянемо способи адресації і адресну арифметику.

При 32-розрядному програмуванні логічна (віртуальна) адреса визначається так:

селектор_сегмента: 32-разрядна_ефективна_адреса.

Ефективна 32-розрядна адреса (Effective Address) дорівнює кількості байтів від початку сегмента (зсуву). У загальному випадку вона обчислюється складанням будь-якої комбінації таких чотирьох адресних елементів:

ефективна_адреса = База + (Індекс * Масштаб) + Зсув,

де *зсув* – числове значення;

база – уміст кожного з реґістрів загального призначення; реґістр, що тут використовується, називається базовим;

індекс – уміст кожного з реґістрів загального призначення, крім ***ESP***; використовуваний тут реґістр називається індексним;

масштаб – константа 2, 4 або 8.

Для отримання ефективної адреси змінної можна використовувати оператор ***OFFSET***:

MOV приймач, OFFSET ім'я_змінної,

або команду ***LEA*** (Load Effective Address):

LEA приймач, ім'я_змінної.

Наприклад,

LEA EBX, a; – в ***EBX*** завантажується адреса змінної ***a***;
MOV EBX, OFFSET a; – аналогічно попередній команді.

У командах використовуються різні способи задання адрес зберігання операндів (способи адресації):

– *реґістрова адресація*. Операнд знаходиться в реґістрі, назва якого зазначається в команді. Наприклад,

SUB EAX, EBX;

– *безпосередня адресація*. Операнд задається в команді, наприклад,

ADD EAX, 5;

MOV CL, 'D';

– *пряма адресація*. Операнд знаходиться в пам'яті за адресою, яка задається в команді:

MOV EDX, DS: a; – за адресою, визначеною парою *сегмент: зсув*; Assembler замінить ім'я на відповідний зсув;

MOV EBX, a; – сегментний реґістр за замовчуванням ***DS***;

– *побічно-реґістрова адресація*. Операнд знаходиться в пам'яті, його адреса – в реґістрі. Наприклад, у команді

ADD EAX, [EBX]

адреса другого операнда знаходиться в реґістрі ***EBX***;

– *базова адресація.* Адреса операнда дорівнює сумі вмісту реґістра і зсуву, наприклад:

MOV EDX, 8 [EBP].

Інші форми запису: ***[EBP + 8]***; або ***[EBP] +8***;

– *індексна адресація.* Адреса операнда дорівнює сумі вмісту реґістра і зсуву. Наприклад, у команді

MOV mass [ESI], AL

адреса першого операнда дорівнює сумі зсуву відповідного імені ***mass*** і вмісту реґістру ***ESI***;

– *базово-індексна адресація.* Адреса операнда обчислюється як сума вмісту базового й індексного реґістрів, наприклад:

MUL [EBX] [ESI]; або ***[EBX + ESI]***;

– *базово-індексна адресація зі зсувом.* Адреса операнда обчислюється як сума зсуву і вмісту базового й індексного реґістрів, наприклад:

MUL 4 [EBX] [ESI]; або ***[EBX + ESI + 4], [EBX] [ESI] + 4, ...***;

– *індексна адресація з масштабуванням.* Адреса операнда дорівнює сумі зсуву і вмісту індексного реґістра, помноженого на масштаб (1, 2, 4 або 8), наприклад:

MOV EAX, mas [ECX * 2];

– *базово-індексна адресація з масштабуванням.* Адреса операнда дорівнює сумі вмісту базового реґістра і вмісту індексного реґістра, помноженого на масштаб, наприклад:

ADD EAX, [EBX + ESI * 4];

– *базово-індексна адресація зі зсувом і масштабуванням.* Найповніша схема адресації, в якій для обчислення адреси використовуються база, індекс, масштаб і зсув.

Розглянемо одновимірні і двовимірні масиви.

Елементи одновимірного масиву розташовуються в пам'яті комп'ютера послідовно. Доступ до елемента масиву зазвичай здійснюється операцією індексування, яку можна моделювати, знаючи початкову адресу масиву і розмір його елемента в байтах. Тоді адреса i-го (i = 0,1, ...) елемента одновимірного масиву дорівнює

початкова_адреса + (i * розмір_елемента).

Елементи двовимірного масиву розташовуються в пам'яті комп'ютера послідовно. У мові **C** двовимірні масиви розташовуються по рядках. Тоді для масиву розмірністю $n \times m$ елемент у рядку i (0<i<n) і стовпці j (0<j<m) має адресу

початкова_адреса + (i * m * розмір_елемента) +(j * розмір_елемента).

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання 1. Знайти найбільше значення матриці.

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

void main()
{
	//creating the necessary variables and array
	const int N = 100;		//declaration variable const "N", the initial dimension of
the array
	int arr[N];			// declaration array "arr[N]"
	int res = 0;			//declaration variable "res", to store the result
	int max = 0;			//declaration variable "max", to store the maximum array
element
	int n = 0;			//declaration variable "n", the real dimension of the array
	int l = 0;			//declaration variable "l", the array counter

	//read variable values from file
	ifstream file_in("filein.txt");	//creating object for input from file and open file
"Input.txt" for reading value "n" and array
	file_in >> n;			//read variable "n" from file
	l = n - 1;			//the array counter
	for (int i = 0; i < n; i++)//cycle for reading array elements
	{				//start of cycle
		file_in >> arr[i];		//read array value from file
	}				//end of cycle
	file_in.close();			//close the file after reading the variable and array

	_asm	//using Assembler language
	{	//the beginning of the assembler insert
		mov ecx, l		//record the counter to register
		xor ebx, ebx	//setting the register ebx to 0
		mov edx, arr[ebx]//the address of the first element of the array
		add ebx, 4	//transition to the next element of the array

		_begin :		//start loop
		cmp edx, arr[ebx]	//comparing an array element with the following
			jge _greater_equal//if the first operand is greater or equal, go to the
label
			jl _lower	//if the first operand is smaller, go to the label

			_greater_equal :	//the beginning of the program block corresponding to
the label
		mov max, edx		//record register edx to max
			jmp __exit	//go to the label __exit

			_lower :		//the beginning of the program block corresponding to
the label

		mov	edx, arr[ebx]	//record arr[ebx] to register edx
			mov max, edx	//record register edx to max
			jmp __exit	//go to the label __exit

			__exit :		//the beginning of the program block corresponding to
the label
		add ebx, 4		//add 4 to the register ebx, that is, the transition to the
next element of the array
			loop _begin	//go to the label by loop until the counter is 0
			mov ecx, max	//record max to register ecx
			mov res, ecx	//record register ecx to res
	}			//end of assembler insert
	//output to the file
	ofstream out("fileout.txt"); //creating object for write results to file opening file
"Output.txt" to record results
	out << "Array:\t";		//record to file phrases "Array:" and use tabulation for
indent
	cout << "Array:\t";		//record  to console phrases "Array:" and use tabulation for
indent
	for (int i = 0; i < n; i++) //cycle for write array elements
```

```
	{                              //start of cycle
		out << arr[i] << "  ";  //record to file the value of the array element with
number "i" and indent
		cout << arr[i] << "  "; //record to console file the value of the array element
with number "i" and indent
	}                              //end of cycle

	out << "\nMax:\t" << res << endl;  //transition to a new line, record to file the
variable "res", maximum array element
	cout << "\nMax:\t" << res << endl; //transition to a new line, record to console value
the variable "res", maximum array element
	out.close();                       //close the file after record the variables
	system("pause");                   //waiting to press the button "enter", for the end of
the program
}
```

Вхідний файл показано на рисунку 10.1, а результат роботи програми – на рисунках 10.2 і 10.3.

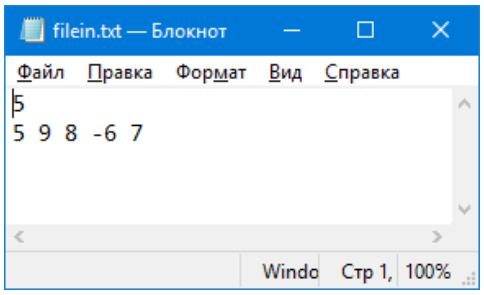


Рисунок 10.1 – Вхідний файл

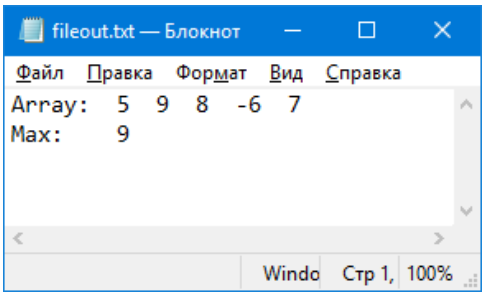


Рисунок 10.2 – Файл результату

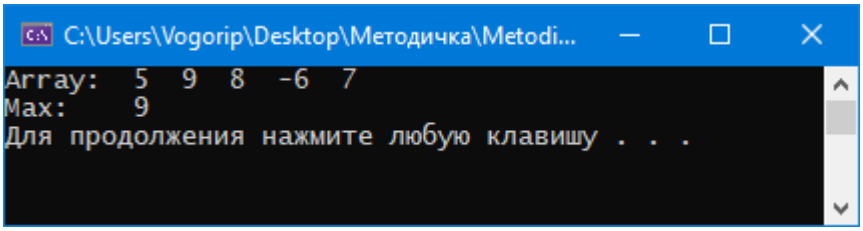


Рисунок 10.3 – Результат роботи програми в консолі

Завдання 2. Підрахувати кількість негативних елементів.

Лістинг програми такий:

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

void main()
{
	int rowCount = 4,                        //number of row
		colCount = 4;                        //number of column
	int arr[100] = { -3,4,7,-5,2,1,-8,3,14,11,16,9,-11,-1,10,3 };//source array
	int neg;                                          //variable to hold the number of
negative element
	_asm
	{                                                      //The beginning of the assembly
insert
```

```
            pushad                                  //Push the values of all 32-bit general-
purpose registers to the stack
            xor edx, edx                    //for storage number of negative element
            lea eax, arr                    //eax <- pointer(arr
            mov ecx, colCount               //ecx <- colCount
            _col_loop :                             //label
            push ecx                                //save pointer
                  mov ecx, rowCount         //ecx <- rowCount
                  _row_loop :                               //label
            push eax                                //save pointer
                  cmp[eax], 0                       //comparing elements to find negative
                  jl _lower                         //jump to label "_lower" if the element
< 0
                  jmp _step             //jump to label "step"
                  _lower :                                  //label
            inc edx                                 //edx + 1
                  jmp _step                         //jump to label "step"
                  _step :                                   //label
            pop eax                                 //return pointer
                  add eax, 4                        //jump to the next row element
                  loop _row_loop                    //until the counter is zero go to label
"_row_loop"
                  pop ecx                                   //return pointer
                  loop _col_loop                    //until the counter is zero go to label
"_column_loop"
                  mov neg, edx              //neg <- edx
                  popad                 //Extracting from the stack the values of all 32-
bit general-purpose registers
      }

      cout << "Array: " << endl;                                          //output to the
console text "Array:"
      for (int i = 0; i < rowCount; i++, cout << endl)                                ///
            for (int j = 0; j < colCount; j++)
///output to the console sourece array
                  cout << setw(4) << arr[i + j * rowCount] << "  ";              ///

      cout << "\nThe number of negative elements in the array: " << neg << endl;//output to
the console text "The number of negative elements in the array:" and number of negative
element


      ofstream out_file("fileout.txt");

      out_file << "Array: " << endl;
      //output to the file text "Array:"

      for (int i = 0; i < rowCount; i++, out_file << endl)                          ///
            for (int j = 0; j < colCount; j++)
      ///output to the file sourece array
                  out_file << setw(4) << arr[i + j * rowCount] << "  ";              ///
      out_file << "\nThe number of negative elements in the array: " << neg;  //output to the
file text "Minimal array element:" and number of negative element

      out_file.close();
            //file output stream close

      system("pause");
//system call for pause command waiting for any input
}
```

Результат роботи програми показано на рисунках 10.4 і 10.5.

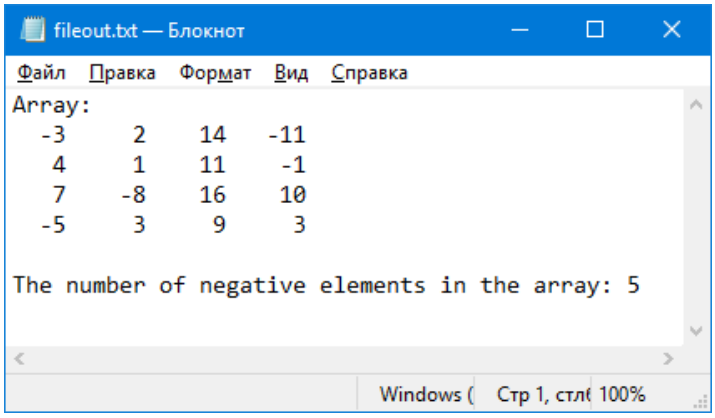


Рисунок 10.4 – Файл результату

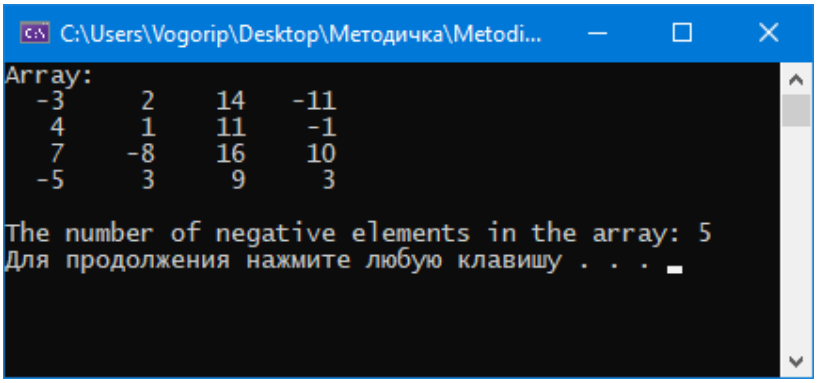


Рисунок 10.5 – Результат роботи програми в консолі

Варіанти завдань

Завдання 1. Дано одновимірний масив з 25 елементів. У таблиці 10.1 наведено варіанти оброблення цього масиву.

Завдання 2. Дано двовимірний масив з 25 елементів (матриця розмірністю *n*-рядків і *m*-стовпців: $n = 5$, $m = 5$). У таблиці 10.2 наведено варіанти оброблення цього масиву.

Таблиця 10.1 – Варіанти завдання 1

Номер варіанта	Завдання
1	Знайти середнє арифметичне парних елементів масиву
2	Знайти середнє арифметичне непарних елементів масиву
3	Знайти середнє арифметичне усіх негативних елементів масиву

Продовження таблиці 10.1

Номер варіанта	Завдання
4	Знайти середнє арифметичне усіх позитивних елементів масиву
5	Знайти суму модулів негативних елементів масиву
6	Знайти добуток усіх позитивних елементів масиву
7	Знайти добуток усіх негативних елементів масиву
8	Знайти найбільше значення масиву
9	Знайти найменше значення масиву
10	Знайти кількість нульових елементів масиву
11	Знайти кількість негативних елементів масиву
12	Знайти кількість позитивних елементів масиву
13	Знайти суму найбільшого і найменшого елементів
14	Знайти різницю між найбільшим і найменшим елементами
15	Знайти добуток найбільшого і найменшого елементів
16	Замінити нулями всі негативні елементи масиву
17	Замінити нулями всі позитивні елементи масиву
18	Замінити одиницями всі нульові елементи масиву
19	Замінити всі негативні елементи масиву їх квадратами
20	Замінити всі позитивні елементи масиву їх кубами
21	Замінити негативні елементи масиву їх модулями
22	Замінити позитивні елементи масиву на їх протилежні негативні
23	Замінити максимальним елементом всі нульові елементи масиву
24	Замінити мінімальним елементом всі нульові елементи масиву
25	Отримати новий масив шляхом складання всіх елементів вихідного масиву з його найбільшим за модулем елементом
26	Отримати новий масив шляхом складання всіх елементів вихідного масиву з його найменшим за модулем елементом
27	Отримати новий масив шляхом віднімання всіх елементів вихідного масиву від його найбільшого за модулем елемента
28	Отримати новий масив шляхом віднімання всіх елементів вихідного масиву від його найменшого за модулем елемента
29	Отримати новий масив шляхом ділення всіх елементів вихідного масиву на його найменший за модулем елемент

Продовження таблиці 10.1

Номер варіанта	Завдання
30	Отримати новий масив шляхом ділення всіх елементів вихідного масиву на його найменший за модулем елемент
31	Знайти середнє арифметичне усіх елементів масиву за модулем
32	Знайти різницю між середнім арифметичним усіх елементів масиву за модулем і найбільшим елементом
33	Знайти різницю між середнім арифметичним усіх елементів масиву за модулем і найменшим елементом
34	Знайти суму між середнім арифметичним усіх негативних елементів масиву і найменшим елементом
35	Знайти суму між середнім арифметичним усіх позитивних елементів масиву і найбільшим елементом
36	Замінити всі негативні елементи масиву їх кубами
37	Замінити всі позитивні елементи масиву їх квадратами
38	Знайти найбільше значення масиву з елементів за модулем
39	Знайти найменше значення масиву з елементів за модулем
40	Знайти середнє арифметичне за модулем усіх негативних елементів масиву
41	Знайти середнє арифметичне квадратів усіх позитивних елементів масиву
42	Знайти середнє арифметичне квадратів усіх негативних елементів масиву
43	Знайти суму найбільшого за модулем і найменшого за модулем елемента
44	Знайти модуль добутку всіх позитивних елементів масиву
45	Знайти модуль добутку всіх негативних елементів масиву
46	Замінити одиницями всі негативні елементи масиву
47	Замінити одиницями всі позитивні елементи масиву
48	Замінити всі нульові елементи масиву на 100
49	Знайти частку від ділення максимального елемента на мінімальний
50	Знайти частку за модулем від ділення суми позитивних елементів на суму негативних

Таблиця 10.2 – Варіанти завдання 2

Номер варіанта	Завдання
1	Знайти середнє арифметичне елементів кожного з парних стовпців матриці і сформувати з них вектор
2	Знайти середнє арифметичне елементів кожного з непарних стовпців матриці і сформувати з них вектор
3	Знайти середнє арифметичне елементів кожного з рядків матриці і сформувати з них вектор
4	Знайти середнє арифметичне елементів кожного з парних рядків матриці і сформувати з них вектор
5	Знайти середнє арифметичне елементів кожного з непарних рядків матриці і сформувати з них вектор
6	Знайти середнє арифметичне усіх негативних елементів матриці
7	Знайти середнє арифметичне усіх позитивних елементів матриці
8	Знайти характеристики кожного рядку матриці (суму додатних парних елементів у кожному рядку) і занести їх у вихідний вектор
9	Знайти характеристики кожного стовпця матриці (суму модулів негативних непарних елементів у кожному стовпці) і занести їх у вихідний вектор
10	Знайти суму і добуток усіх позитивних елементів матриці
11	Знайти суму і добуток усіх негативних елементів матриці
12	Знайти суму всіх позитивних і добуток усіх негативних елементів
13	Знайти суму всіх негативних і добуток усіх позитивних елементів
14	Знайти суму всіх елементів і замінити нею усі діагональні елементи матриці
15	Знайти добуток всіх елементів і замінити ним усі діагональні елементи матриці
16	Знайти суму всіх позитивних елементів і замінити нею усі діагональні елементи матриці
17	Знайти добуток всіх позитивних елементів і замінити ним усі діагональні елементи матриці
18	Знайти суму всіх негативних елементів і замінити нею усі діагональні елементи матриці

Продовження таблиці 10.2

Номер варіанта	Завдання
19	Знайти добуток всіх негативних елементів і замінити ним усі діагональні елементи матриці
20	Знайти суму найбільших елементів кожного рядка матриці
21	Знайти суму найменших елементів кожного рядка матриці
22	Знайти добуток найбільших елементів кожного рядка матриці
23	Знайти добуток найменших елементів кожного рядка матриці
24	Знайти суму найбільших елементів кожного стовпця матриці
25	Знайти суму найменших елементів кожного стовпця матриці
26	Знайти добуток найбільших елементів кожного стовпця матриці
27	Знайти добуток найменших елементів кожного стовпця матриці
28	Знайти кількість позитивних елементів у кожному рядку матриці і сформувати з них вектор
29	Знайти кількість негативних елементів у кожному рядку матриці і сформувати з них вектор
30	Знайти кількість позитивних елементів у кожному стовпці матриці і сформувати з них вектор
31	Знайти кількість негативних елементів у кожному рядку матриці і сформувати з них вектор
32	Отримати нову матрицю шляхом ділення всіх елементів вихідної матриці на її найменший за модулем елемент
33	Отримати нову матрицю шляхом ділення всіх елементів вихідної матриці на її найменший за модулем елемент
34	Отримати нову матрицю шляхом множення всіх елементів вихідної матриці на її найбільший за модулем елемент
35	Отримати нову матрицю шляхом множення всіх елементів вихідної матриці на її найменший за модулем елемент
36	Отримати нову матрицю шляхом складання всіх елементів вихідної матриці з її найбільшим за модулем елементом
37	Отримати нову матрицю шляхом складання всіх елементів вихідної матриці з її найменшим за модулем елементом
38	Отримати нову матрицю шляхом віднімання від усіх елементів матриці її найбільшого за модулем елемента
39	Отримати нову матрицю шляхом віднімання від усіх елементів вихідної матриці її найменшого за модулем елемента

Продовження таблиці 10.2

Номер варіанта	Завдання
40	Замінити нулями всі елементи матриці, розташовані на головній діагоналі і вище неї
41	Замінити нулями всі елементи матриці, розташовані на головній діагоналі і нижче неї
42	Сформувати вектор з суми елементів рядків і знайти їх середнє арифметичне
43	Сформувати вектор з суми елементів стовпців і знайти їх середнє арифметичне
44	Сформувати вектор з добутків елементів рядків і знайти їх середнє арифметичне
45	Сформувати вектор з добутків елементів стовпців і знайти їх середнє арифметичне
46	Сформувати вектор з найменших значень елементів рядків і знайти їх середнє арифметичне
47	Сформувати вектор з найменших значень елементів стовпців і знайти їх середнє арифметичне
48	Сформувати вектор з найбільших значень елементів рядків і знайти їх середнє арифметичне
49	Сформувати вектор з найбільших значень елементів стовпців і знайти їх середнє арифметичне
50	Сформувати вектор з різниць найбільших і найменших значень елементів рядків

Лабораторна робота № 11

ПРОЦЕДУРИ, ЇХ ВИКЛИК, ПЕРЕДАЧА ПАРАМЕТРІВ

Мета роботи: вивчити поняття стеку і команди для роботи з ним; ознайомитися з організацією процедур у мові Assembler, їх викликом і передачею параметрів для роботи.

Завдання

1. Ознайомитися з необхідними командами мови Assembler і особливостями їх використання.

2. Проаналізувати завдання відповідно до варіанта. Оцінити область визначення змінних, визначити типи даних і виняткові ситуації, які можуть

виникнути. Скласти алгоритм виконання необхідних дій, визначити порядок їх використання.

3. Написати мовою Assembler програму, що виконує поставлене завдання. Результати обчислень вивести на екран у десятковій і двійковій системах числення. Засвоїти включення команд мови Assembler у програму, написану мовою високого рівня.

4. Протестувати програму на різних вхідних даних.

5. Скласти звіт, який повинен містити тему, мету роботи, завдання відповідного варіанта, опис виняткових ситуацій та особливостей виконання завдання, лістинг програмного коду.

Теоретичні відомості

Часто програмі потрібно тимчасово запам'ятати інформацію. Для цього в програмі використовується спеціальний сегмент – сегмент стеку, що називається *стеком*. При розміщенні елементів у стеку відбувається зменшення покажчика вершини стеку, а при вилученні – його збільшення. Тобто стек завжди «зростає» у бік менших адрес пам'яті.

Для роботи зі стеком використовуються реґістри ***SS***, ***ESP*** і ***EBP***. Уміст ***SS*** є базою стеку. У ***ESP*** зберігається зсув вершини стеку. Спочатку ***ESP*** ініціалізується найбільшим зсувом, якого може досягати стек, потім змінюється операціями включення і вилучення. Реґістр ***EBP*** зазвичай використовується для звернень до елементів стеку.

Нижче розглянуто деякі команди роботи зі стеком.

Команда збереження даних у стеку:

PUSH джерело.

Джерелом можуть бути реґістр, сегментний реґістр або змінна. ***ESP*** зменшується на розмір джерела в байтах (2 або 4), і вміст джерела розміщується в пам'яті за адресою ***SS: ESP***.

Команда вилучення даних зі стеку:

POP приймач.

Ця команда поміщує в приймач слово або подвійне слово, яке знаходиться у вершині стеку, збільшуючи ***ESP*** на 2 або 4 відповідно. Приймачем можуть бути реґістр загального призначення, сегментний реґістр (крім ***CS***) або змінна. Якщо в ролі приймача виступає операнд, який використовує ***ESP*** для непрямої адресації, команда ***POP*** обчислює адресу операнда вже після того, як вона збільшує ***ESP*.**

Команда ***PUSH*** часто використовується в парі з ***POP***.

Копіювання одного сегментного реґістра в інший (що виконується однією командою ***MOV***), можна реалізувати так:

PUSH DS
POP ES.

Для тимчасового зберігання даних можна виконати таке:

PUSH EAX
; команди зміни EAX
POP EAX.

Команди ***PUSHA*** і ***PUSHAD*** поміщують у стек всі реґістри загального призначення. ***PUSHA*** має в стеці реґістри в такому порядку: ***AX***, ***CX***, ***DX***, ***BX***, ***SP***, ***BP***, ***SI*** і ***DI***. ***PUSHAD*** поміщує в стек ***EAX***, ***ECX***, ***EDX***, ***EBX***, ***ESP***, ***EBP***, ***ESI*** і ***EDI***. У випадку з ***SP*** і ***ESP*** використовується значення, яке знаходилося в реґістрі до початку роботи команди. Команди використовуються в парі з командами ***POPA*** і ***POPAD***, що зчитують ці ж реґістри зі стека у зворотному порядку, це дозволяє писати процедури, які не повинні змінювати значення реґістрів після закінчення своєї роботи. На початку такої підпрограми викликають команду ***PUSHA***, а в кінці – ***POPAУ***.

Зберегти/витягти у/зі стеку реґістр прапорів можна командами ***PUSHF/POPF.***

Традиційно сегмент коду складається з процедур, хоча це і не є обов'язковим. Для оформлення процедури використовується конструкція

Ім’я_процедури PROC [NEAR/FAR]
;тіло процедури
RET
Ім’я_процедури ENDP.

Можна вказати тип процедури ***NEAR*** або ***FAR***, за замовчуванням використовується ***NEAR***. Залежно від типу при виклику процедури по-різному запам'ятовується в стеку адреса повернення. У разі ближньої процедури (***NEAR***) у стек поміщується зсув, а у разі далекої (***FAR***) – повна логічна адреса (база та зсув). Команда ***RET*** витягує зі стеку адресу повернення в реґістр ***EIP***.

Для виклику процедури використовується команда ***CALL***:

CALL ім’я_процедури.

Передача параметрів (за значенням і за адресою) проводиться через реґістри і через стек.

При передачі через реґістри параметри заносяться у заздалегідь зумовлені реґістри, і виклична процедура оперує з ними.

Приклад виконання роботи на C++ з використанням вставок мовою Assembler

Завдання. Створити процедуру складання двох 32-бітних цілих. Використовувати передачу параметрів через стек. Стек очищує виклична процедура.

Лістинг програми такий:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   // creating the necessary variables
    int a, b;
    int res;

   // read variable values from file
    ifstream file_in;// creating object for input from file
    file_in.open("filein.txt");// open file "filein.txt" for reading values "a" and "b"
    file_in >> a;// read variable "a"
    file_in >> b;// variable "mask"
    file_in.close();// close the file after reading the variables

    __asm// using Assembler language
    {
        jmp program//
        sum_num :// start loop
        push ebp
            mov ebp, esp
            mov eax, dword ptr[ebp + 8]
            mov ebx, dword ptr[ebp + 12]
            add eax, ebx
            pop ebp
            ret 8
            program://the beginning of the program block corresponding to the label
        mov eax, a
            push eax
            mov ebx, b
            push ebx
            call sum_num
            mov res, eax
    }// end of assembler insert
   // output to the file
    ofstream file_out;// creating object for write results to file
    file_out.open("fileout.txt");// opening file "fileout.txt" to record results
    file_out << "Res =" << res << endl;// record to file value the variable "Res"
    file_out.close();// close the file after record the variables

   // output to console
    cout << "Res =" << res << endl;// record to console value the variable "Res"

    system("pause");// waiting to press the button "enter", for the end of the program
}
```

Вхідний файл показано на рисунку 11.1, а результати виконання програми – на рисунках 11.2 і 11.3.

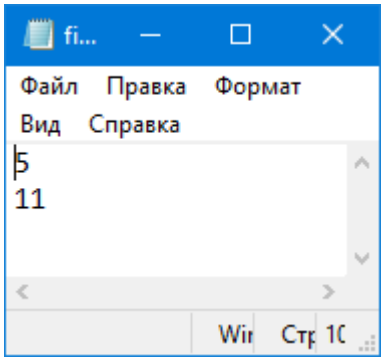


Рисунок 11.1 – Вхідний файл

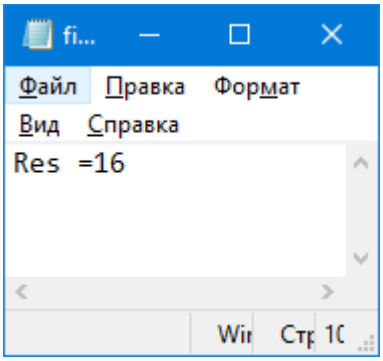


Рисунок 11.2 – Файл результату

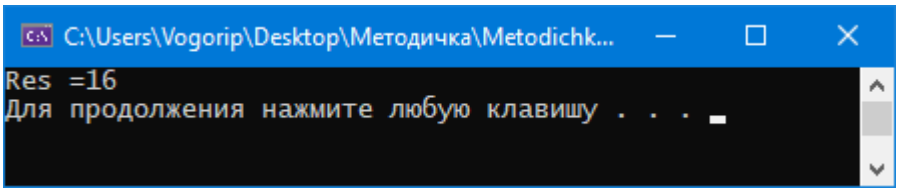


Рисунок 11.3 – Результат роботи програми в консолі

Варіанти завдань

Запрограмувати процедуру згідно з варіантом. Варіанти завдань наведено у таблиці 11.1. У варіантах 1 – 6 потрібно використовувати передачу параметрів через реґістри; у варіантах 7 – 42 – через стек, стек має очищати виклична процедура.

Таблиця 11.1 – Варіанти завдання

Номер варіанта	Завдання
1	Процедура складання двох цілих чисел
2	Процедура віднімання двох цілих чисел
3	Процедура беззнакового множення двох цілих чисел
4	Процедура знакового множення двох цілих чисел
5	Процедура беззнакового ділення двох цілих чисел
6	Процедура знакового ділення двох цілих чисел
7	Процедура складання двох 8-бітних цілих
8	Процедура складання двох 16-бітних цілих
9	Процедура складання двох 32-бітних цілих
10	Процедура складання двох 8-бітних цілих
11	Процедура складання двох 16-бітних цілих

Продовження таблиці 11.1

Номер варіанта	Завдання
12	Процедура складання двох 32-бітних цілих
13	Процедура віднімання двох 8-бітних цілих
14	Процедура віднімання двох 16-бітних цілих
15	Процедура віднімання двох 32-бітних цілих
16	Процедура віднімання двох 8-бітних цілих
17	Процедура віднімання двох 16-бітних цілих
18	Процедура віднімання двох 32-бітних цілих
19	Процедура беззнакового множення двох 8-бітних цілих
20	Процедура беззнакового множення двох 16-бітних цілих
21	Процедура беззнакового множення двох 32-бітних цілих
22	Процедура беззнакового множення двох 8-бітних цілих
23	Процедура беззнакового множення двох 16-бітних цілих
24	Процедура беззнакового множення двох 32-бітних цілих
25	Процедура знакового множення двох 8-бітних цілих
26	Процедура знакового множення двох 16-бітних цілих
27	Процедура знакового множення двох 32-бітних цілих
28	Процедура знакового множення двох 8-бітних цілих
29	Процедура знакового множення двох 16-бітних цілих
30	Процедура знакового множення двох 32-бітних цілих
31	Процедура беззнакового ділення двох 8-бітних цілих
32	Процедура беззнакового ділення двох 16-бітних цілих
33	Процедура беззнакового ділення двох 32-бітних цілих
34	Процедура беззнакового ділення двох 8-бітних цілих
35	Процедура беззнакового ділення двох 16-бітних цілих
36	Процедура беззнакового ділення двох 32-бітних цілих
37	Процедура знакового ділення двох 8-бітних цілих
38	Процедура знакового ділення двох 16-бітних цілих
39	Процедура знакового ділення двох 32-бітних цілих
40	Процедура знакового ділення двох 8-бітних цілих
41	Процедура знакового ділення двох 16-бітних цілих
42	Процедура знакового ділення двох 32-бітних цілих

Домашнє завдання 1

СПІЛЬНЕ ВИКОРИСТАННЯ МОВИ ASSEMBLER І МОВИ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ C++ З ВИКОРИСТАННЯМ WINDOWS FORM

Порядок виконання завдання

1. Проаналізувати завдання, скласти алгоритм його розв'язання із застосуванням команд мови Assembler, визначити необхідні вхідні і вихідні дані.
2. Написати мовою Assembler процедуру, що виконує поставлене завдання.
3. Написати програму мовою C++ з використанням форм, яка вводить значення вхідних параметрів і здійснює перевірку на аномалії, за необхідності видає попереджувальне повідомлення, після чого проводить оброблення масиву мовою Assembler і виводить результат на екран.
4. Протестувати програму на коректних і аномальних вхідних даних.

Зміст звіту

1. Перелік аномалій і допустимих значень вхідних даних.
2. Значення змінних, для яких можуть бути отримані правильні результати.
3. Лістинг програми мовою C++ і Assembler.
4. Результати розрахунку.

Теоретичні відомості

Розглянемо послідовність дій для створення застосунку зі вставками мовою Assembler.

1. Створити проєкт C++ CLR Empty типу (рисунок 1).

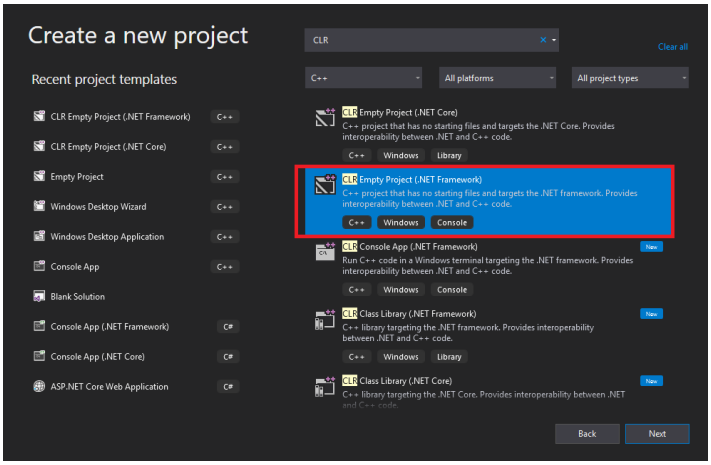


Рисунок 1 – Вікно створення проєкту

2. Додати до проєкту форму C ++ (рисунок 2).

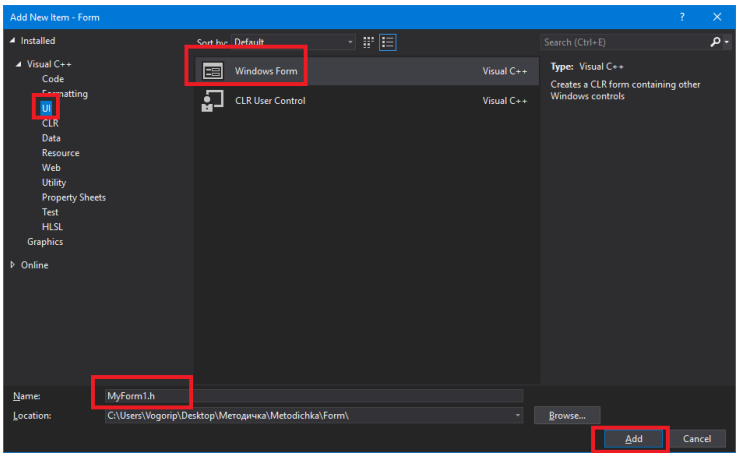


Рисунок 2 – Вікно вставлення форми

3. Відредагувати форму, додати поля для введення даних і кнопки для виклику подій (рисунок 3).

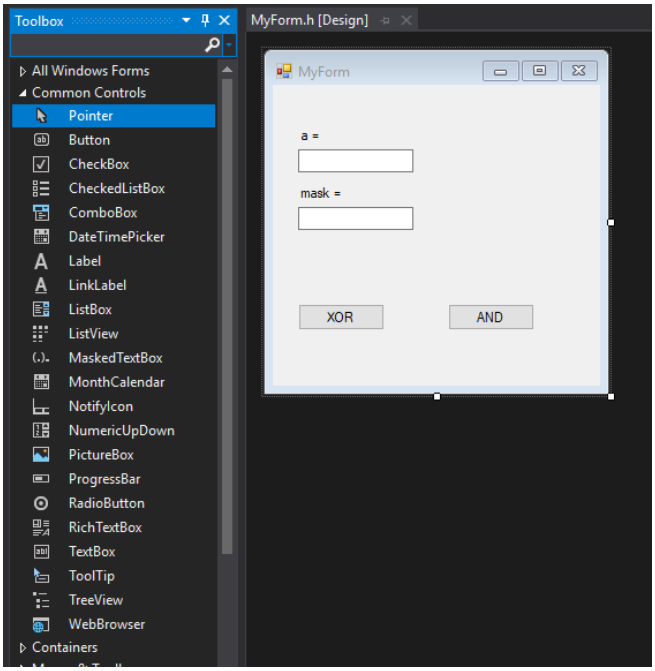


Рисунок 3 – Створення полів і кнопок

4. Написати функції зі вставками мовою Assembler:

```
#pragma once
#pragma intrinsic(strlen)
int asm_xor(int a, int mask) {
	int res;
	__asm
	{
		mov eax, a;// 1372481428, 10100011100 1110 0110 0111 1001 0100
		xor eax, mask;// 1371537412, 10100011100 0000 0000 0000 0000 0100
		mov res, eax;// 944016 , 00000000000 1110 0110 0111 1001 0000
	}
	return res;
}
```

```
#pragma intrinsic(strlen)
int asm_and(int a, int mask) {
	int res;
	__asm {
		mov eax, a;// 1468159445, 10101111000 0010 0101 0101 1101 0101
		and eax, mask;// 1048560 , 00000000000 1111 1111 1111 1111 0000
		mov res, eax;// 153040 , 00000000000 0010 0101 0101 1101 0000
	}
	return res;
      }
```

5. Запрограмувати події натискання кнопки з викликом функцій:

```
#pragma endregion
	private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
		Int64 a, b, c;
		a = Convert::ToInt64(this->textBox1->Text);
		b = Convert::ToInt64(this->textBox2->Text);
		c = asm_xor(a, b);
		this->textBox1->Text = Convert::ToString(c, 2);
		this->textBox2->Text = Convert::ToString(a, 2);
		MessageBox::Show("res = " + c);
	}
	private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e) {
		Int64 a, b, c;
		a = Convert::ToInt64(this->textBox1->Text);
		b = Convert::ToInt64(this->textBox2->Text);
		c = asm_and(a, b);
		this->textBox1->Text = Convert::ToString(c, 2);
		this->textBox2->Text = Convert::ToString(a, 2);
		MessageBox::Show("res = " + c);
	}
```

Результат роботи застосунку показано на рисунку 4.

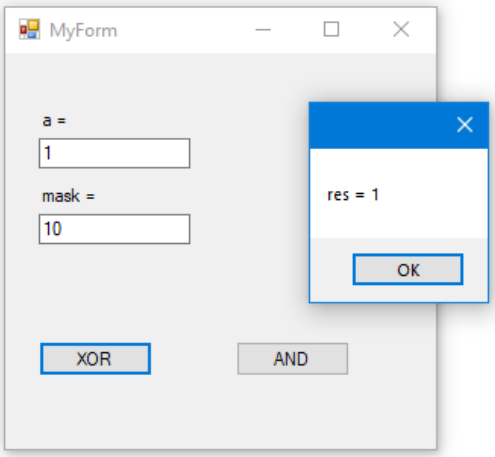


Рисунок 4 – Виконання застосунку

Варіанти завдань

Виконати завдання з лабораторної роботи № 11, використовуючи розглянутий алгоритм дій.

Домашне завдання 2

СПІЛЬНЕ ВИКОРИСТАННЯ МОВИ ASSEMBLER І МОВИ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ C#

Порядок виконання завдання

1. Проаналізувати завдання, скласти алгоритм його розв'язання із застосуванням команд мови Assembler, визначити необхідні вхідні і вихідні дані.

2. Написати мовою Assembler процедуру, що виконує поставлене завдання.

3. Написати програму мовою C#, яка вводить значення вхідних параметрів і здійснює перевірку на аномалії, за необхідності видає попереджувальне повідомлення, після чого проводить оброблення масиву мовою Assembleri і виводить результат на екран.

4. Протестувати програму на коректних і аномальних вхідних даних.

Зміст звіту

1. Перелік аномалій і допустимих значень вхідних даних.

2. Значення змінних, для яких можуть бути отримані правильні результати.

3. Лістинг програми мовами C# і Assembler.

4. Результати розрахунку.

Теоретичні відомості

Розглянемо послідовність дій для створення застосунку.

1. Створити проєкт C# консольного типу (рисунок 5).

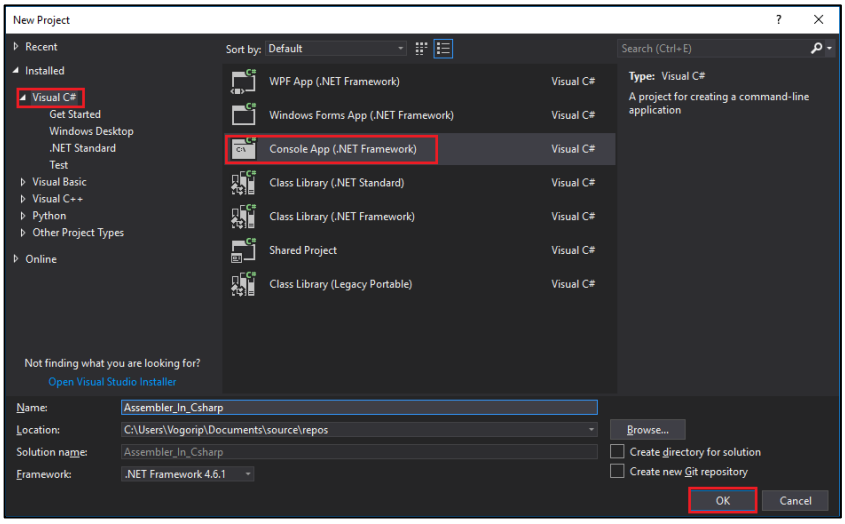
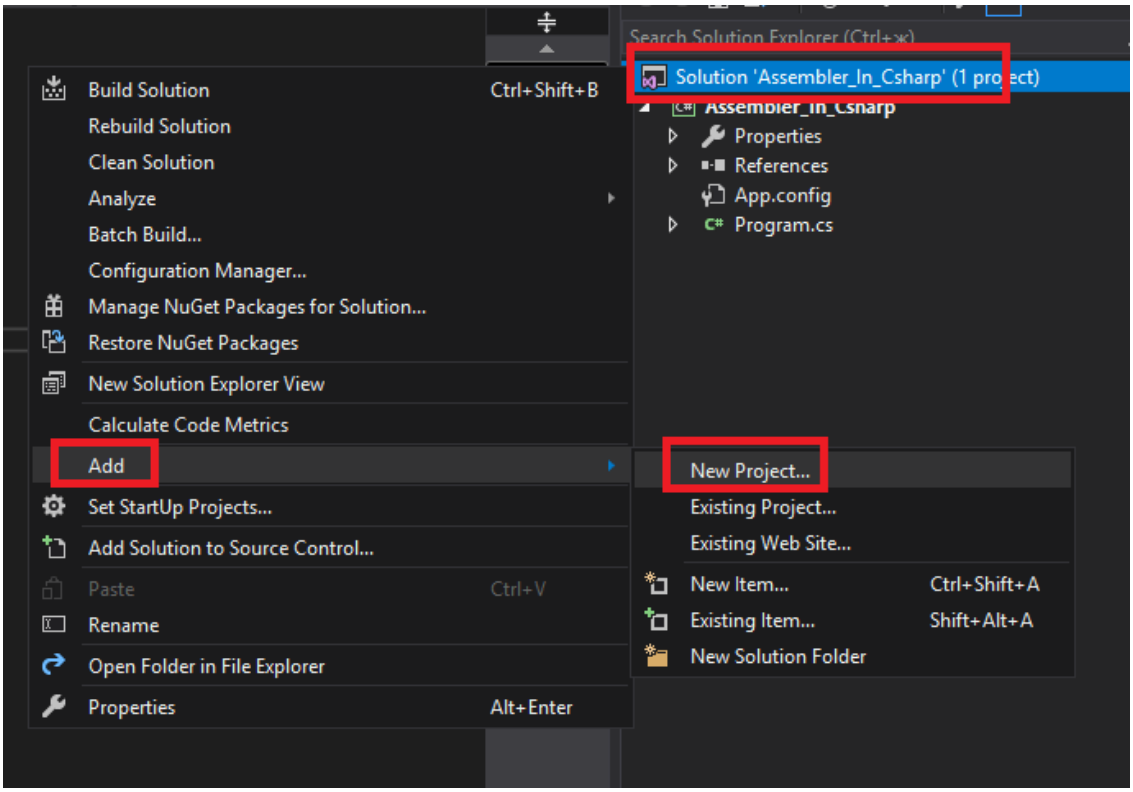
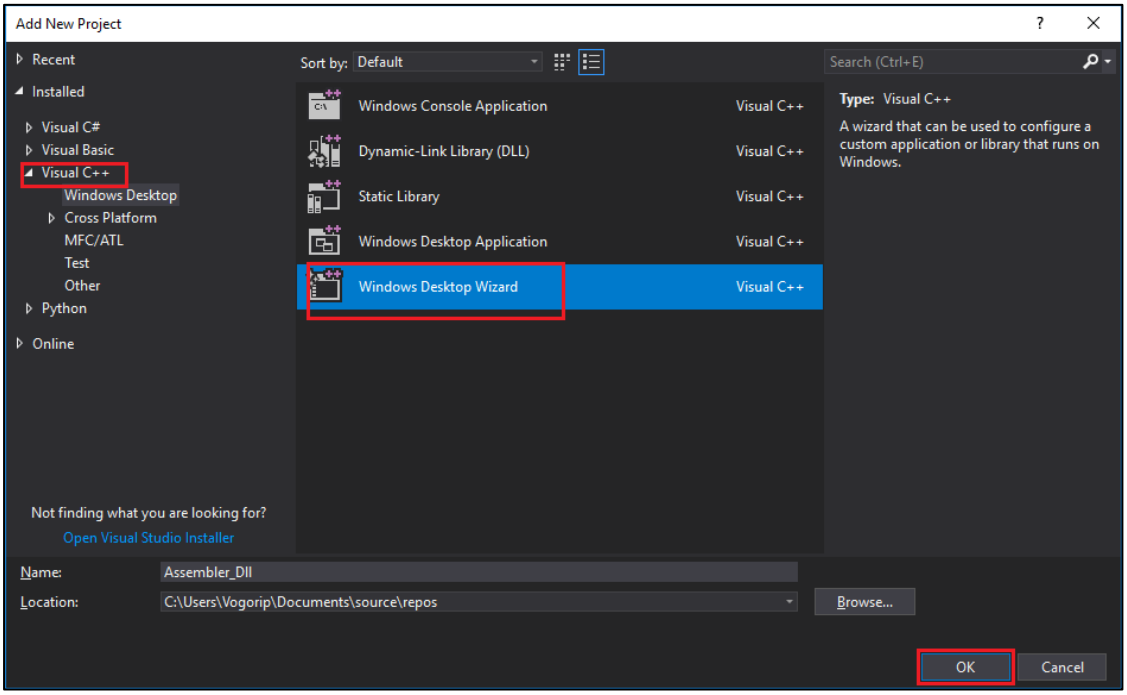


Рисунок 5 – Створення проєкту консольного типу

2. Додати у Solution ще один проєкт, тепер C ++ (рисунок 6, а і 6, б).



а



б

Рисунок 6 – Вставка проєкту C++

3. Вибрати тип DLL і встановити прапорець *Empty Project* (рисунок 7).

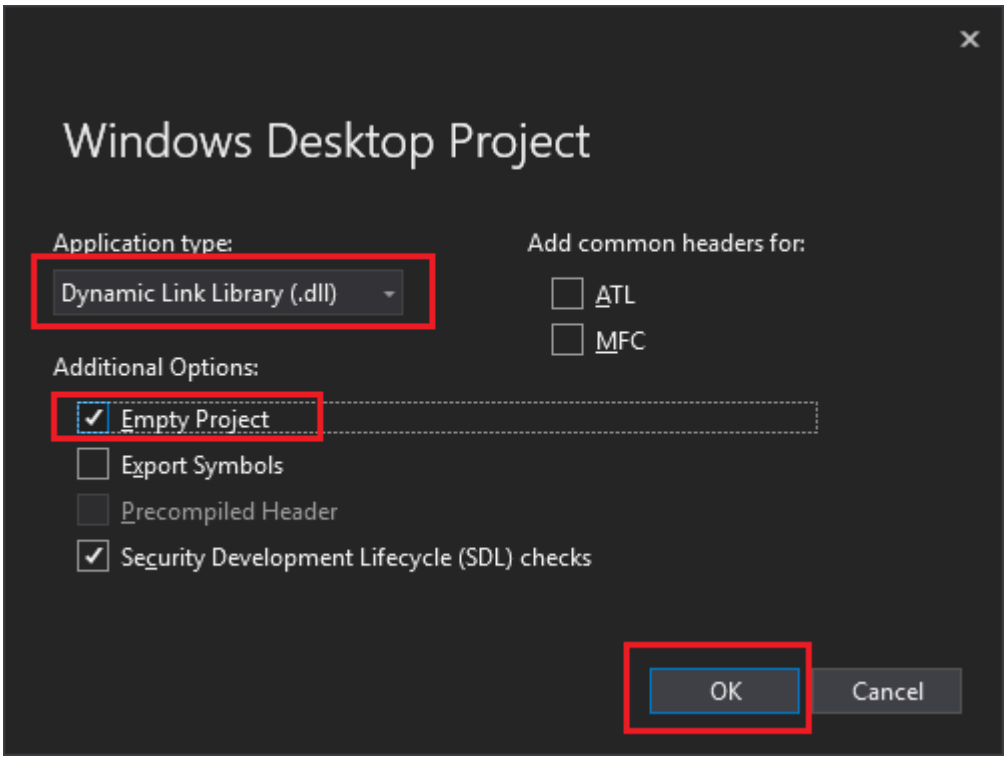


Рисунок 7 – Настроювання проєкту

4. У проєкт C++ додати файли *main.cpp* (рисунок 8) і *main.def* (рисунок 9).

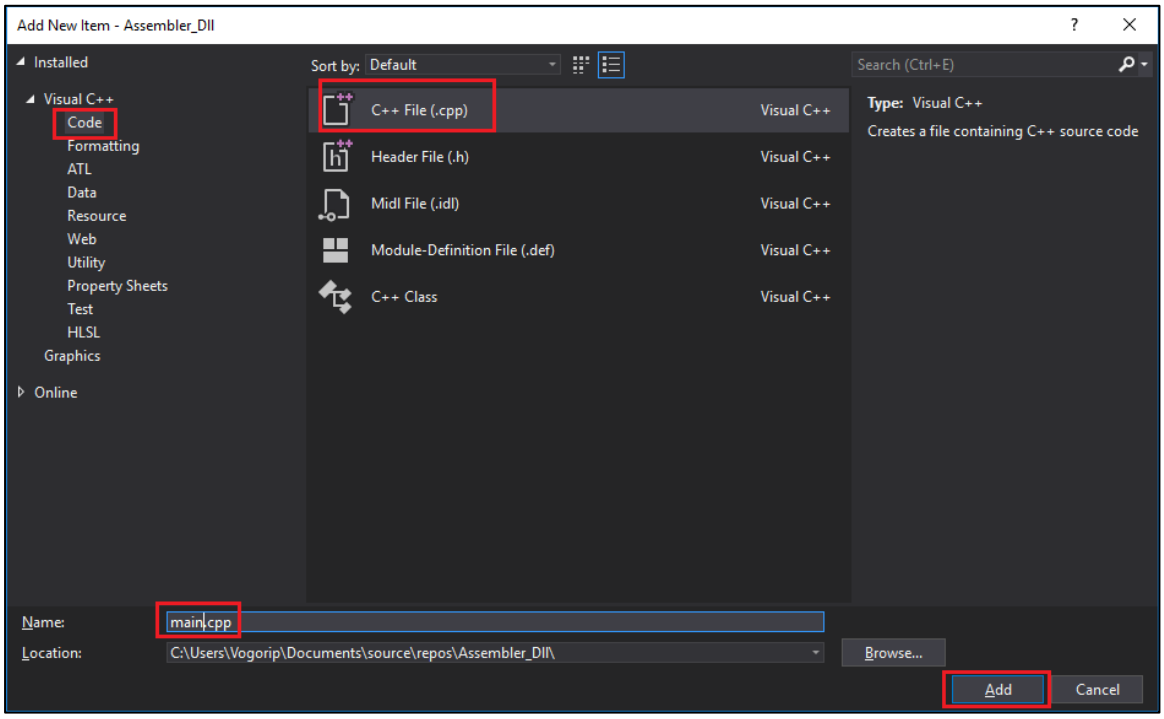


Рисунок 8 – Додавання файлу *main.cpp*

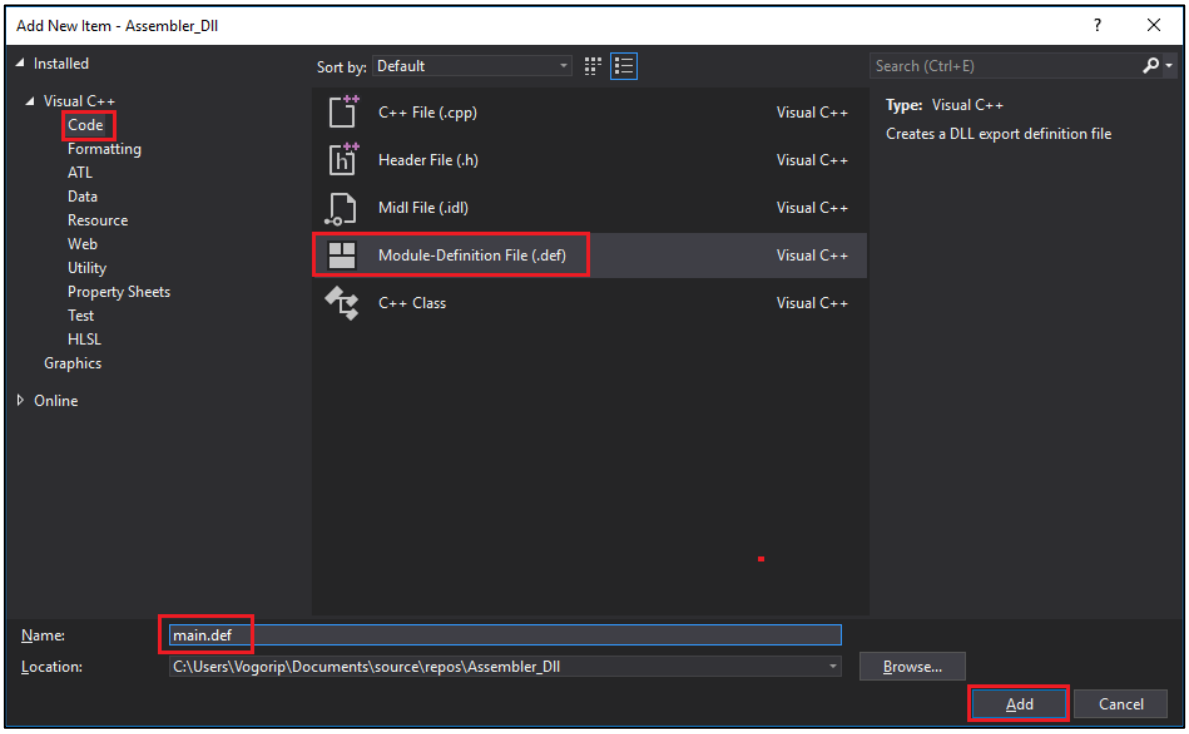


Рисунок 9 – Додавання файлу *main.def*

5. У файлі *main.cpp* потрібно написати функції зі вставкою мовою Assembler для бітових операцій ***XOR*** і ***AND***. Перед кожною функцією необхідно додати модифікатор *__declspec (dllexport)*, який дозволить експортувати функцію з бібліотеки DLL для її використання в інших додатках.

Лістинг функцій такий:

```
__declspec(dllexport)
int asm_xor(int a, int mask)
{
	int res;
	// Xor operation for register into mask
	_asm
	{
		mov eax, a;
		xor eax, mask;
		mov res, eax;
	}
	return res;
}
__declspec(dllexport)
int asm_and(int a, int mask)
{
	int res;
	// And operation for register into mask
	_asm
	{
		mov eax, a;
		and eax, mask;
		mov res, eax;
	}
	return res;
}
```

6. У файлі *main.def* необхідно вказати назву бібліотеки DLL, ключове слово EXPORTS, а після нього – назви функцій, експорт яких необхідно дозволити:

```
LIBRARY "Assembler_Dll"
EXPORTS
asm_xor
asm_and
```

7. Побудувати рішення (Ctrl + Shift + B). У результаті у папці з проєктом буде створено файл *Assembler_Dll.dll*. У папці *Debug* буде знаходитися створений dll-файл (рисунок 10).

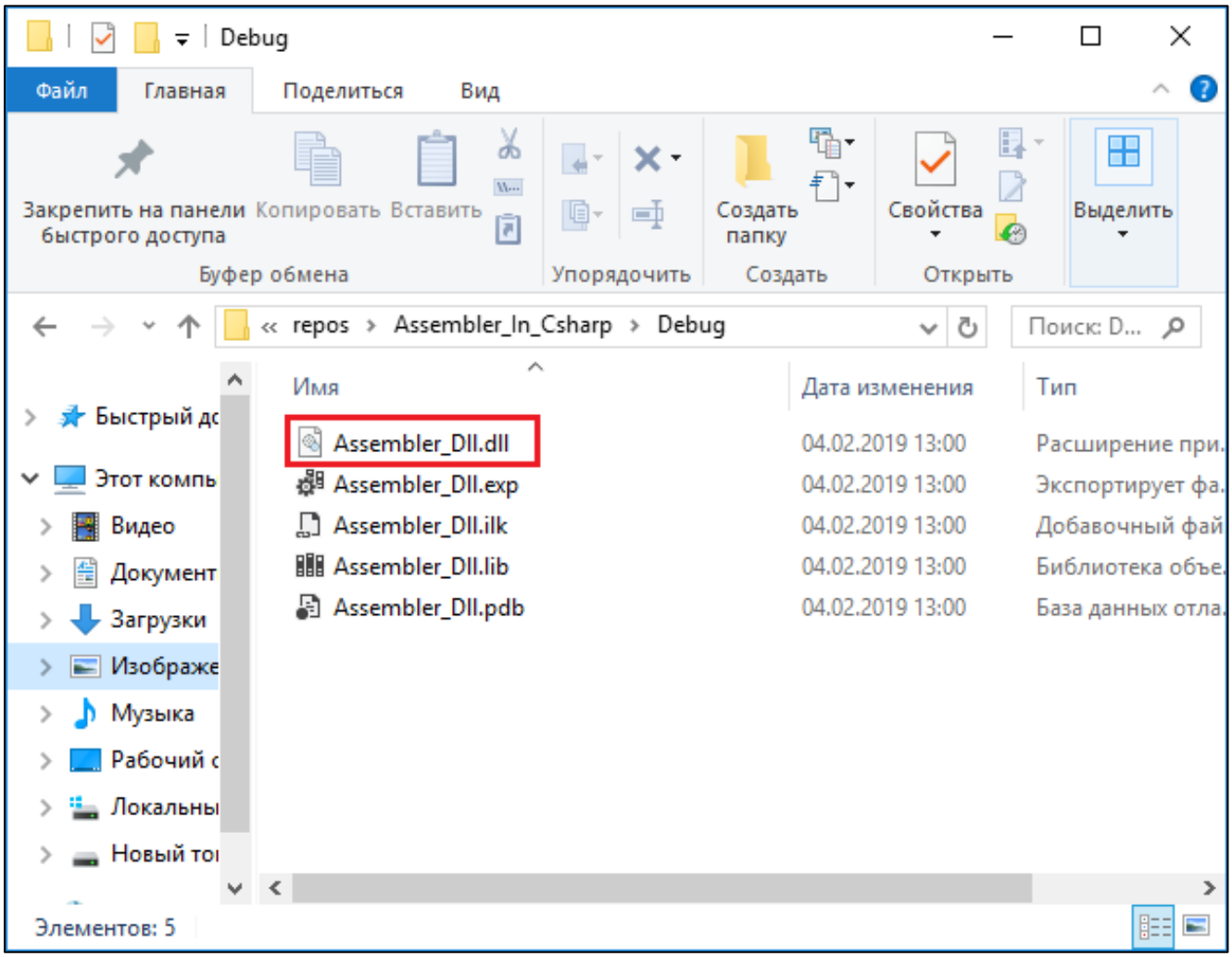


Рисунок 10 – Уміст папки *Debug*

8. Помістити у папку проєкту C# *Debug* файл *Assembler_Dll.dll* (рисунок 11).

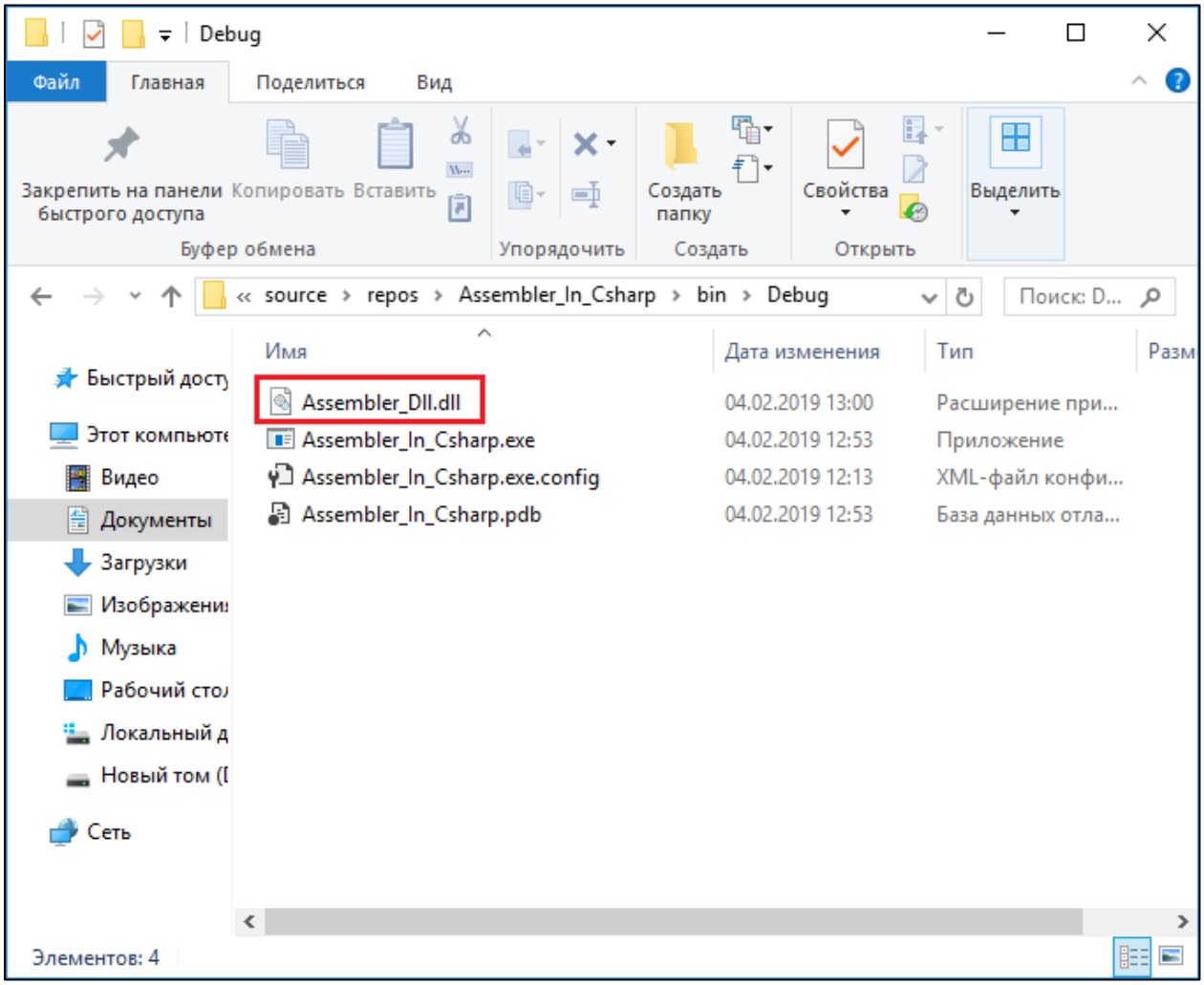


Рисунок 11 – Уміст папки *bin/Debug*

9. У класі *Program* оголосити методи, які реалізовані ззовні (*.dll*), для цього використати модифікатор *extern* з атрибутом *DllImport*. Також потрібно підключити простір імен *System.Runtime.InteropServices*. У цьому випадку метод повинен бути оголошений як *static*:

```
using System;
using System.Runtime.InteropServices;

namespace Csh
{
    class Program
    {
            //Indicates that the attributed method is exposed by an unmanaged dynamic-link
library (DLL) as a static entry point.
            [DllImport("Assembler_Dll.dll", CallingConvention = CallingConvention.Cdecl)]
            //function prototype declaration xor
            static extern int asm_xor(int a, int mask);
            //Indicates that the attributed method is exposed by an unmanaged dynamic-link
library (DLL) as a static entry point.
            [DllImport("Assembler_Dll.dll", CallingConvention = CallingConvention.Cdecl)]
            //function prototype declaration and
            static extern int asm_and(int a, int mask);
```

```
            static void Main()
            {
                  //creating the necessary variables
                  int n1 = 1372481428, m1 = 1371537412;
                  int n2 = 1468159445, m2 = 1048650;
                  //function call xor
                  int res_xor = asm_xor(n1, m1);
                  //function call and
                  int res_and = asm_and(n1, m1);
                  //output to console
                  Console.WriteLine("Number1 = " + n1 + " Binary =" + Convert.ToString(n1,
2));
                  Console.WriteLine("Mask1 = " + m1 + " Binary =" + Convert.ToString(m1,
2));
                  Console.WriteLine("Res Xor = " + res_xor + " Binary =" +
Convert.ToString(res_xor, 2));
                  Console.WriteLine("Number2 = " + n2 + " Binary =" + Convert.ToString(n2,
2));
                  Console.WriteLine("Mask1 = " + m2 + " Binary =" + Convert.ToString(m2,
2));
                  Console.WriteLine("Res And = " + res_and + " Binary =" +
Convert.ToString(res_and, 2));
                  //waiting to press the button "enter", for the end of the program
                  Console.ReadKey();
            }
      }
}
```

Результат роботи застосунку показано на рисунку 12.

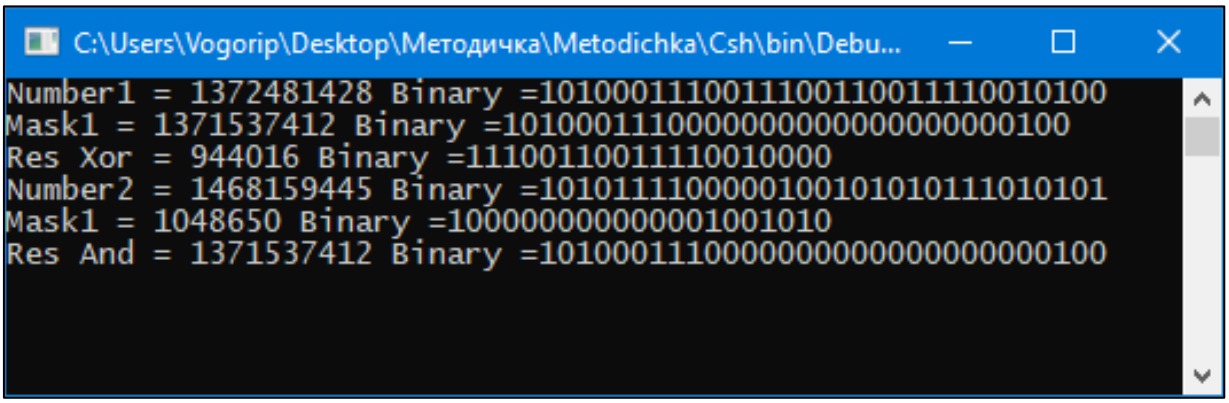


Рисунок 12 – Виведення на екран результатів роботи програми

Варіанти завдань

Виконати завдання з лабораторної роботи № 10, використовуючи розглянутий алгоритм дій.

ДОДАТОК А

Програмна модель архітектури процесорів, режими роботи, набір реґістрів

Будь-яка програма, що виконується, отримує в своє розпорядження певний набір ресурсів процесора. Ці ресурси необхідні для оброблення і зберігання в пам'яті команд і даних програми, а також інформації про поточний стан програми та процесора. Програмну модель процесора в архітектурі IA-32 процесорів Intel становить такий набір ресурсів (рисунок А.1):

– простір пам'яті, що адресується до 232 – 1 байт (4 Гбайт), наприклад для сімейства процесорів Pentium – до 236 – 1 байт (64 Гбайт);

– набір реґістрів для зберігання даних загального призначення;

– набір сегментних реґістрів;

– набір реґістрів стану і управління;

– набір реґістрів пристрою обчислень з плаваючою точкою (співпроцесора);

– набір реґістрів цілочислового MMX-розширення, відображених на реґістри співпроцесора (вперше з'явились в архітектурі процесора Pentium MMX);

– набір реґістрів MMX-розширення з плаваючою точкою (вперше з'явились в архітектурі процесора Pentium III);

– програмний стек – спеціальна інформаційна структура, робота з якою передбачена на рівні машинних команд.

Це основний набір ресурсів. Крім того, до ресурсів, що підтримуються архітектурою IA-32, необхідно віднести порти введення-виведення, лічильники моніторингу продуктивності.

Програмні моделі більш ранніх процесорів (i486, перші Pentium) відрізняються меншим розміром адресного простору оперативної пам'яті (232-1, оскільки розрядність їх адресної шини становить 32 біти) і відсутністю деяких груп реґістрів. Для кожної групи реґістрів у дужках показано, починаючи з якої моделі дана група реґістрів з'явилася у програмній моделі процесорів Intel. Якщо такого позначення немає, то це означає, що дана група реґістрів вже була в процесорах i386 і i486. Що стосується ще більш ранніх процесорів i8086/88, то насправді вони теж повністю подані на схемі, але становлять лише невелику її частину. У програмну модель даних процесорів входять 8-ми і 16-ти розрядні реґістри загального призначення, сегментні реґістри, реґістри FLAGS, IP і адресний простір пам'яті розміром до 1 Мбайт.

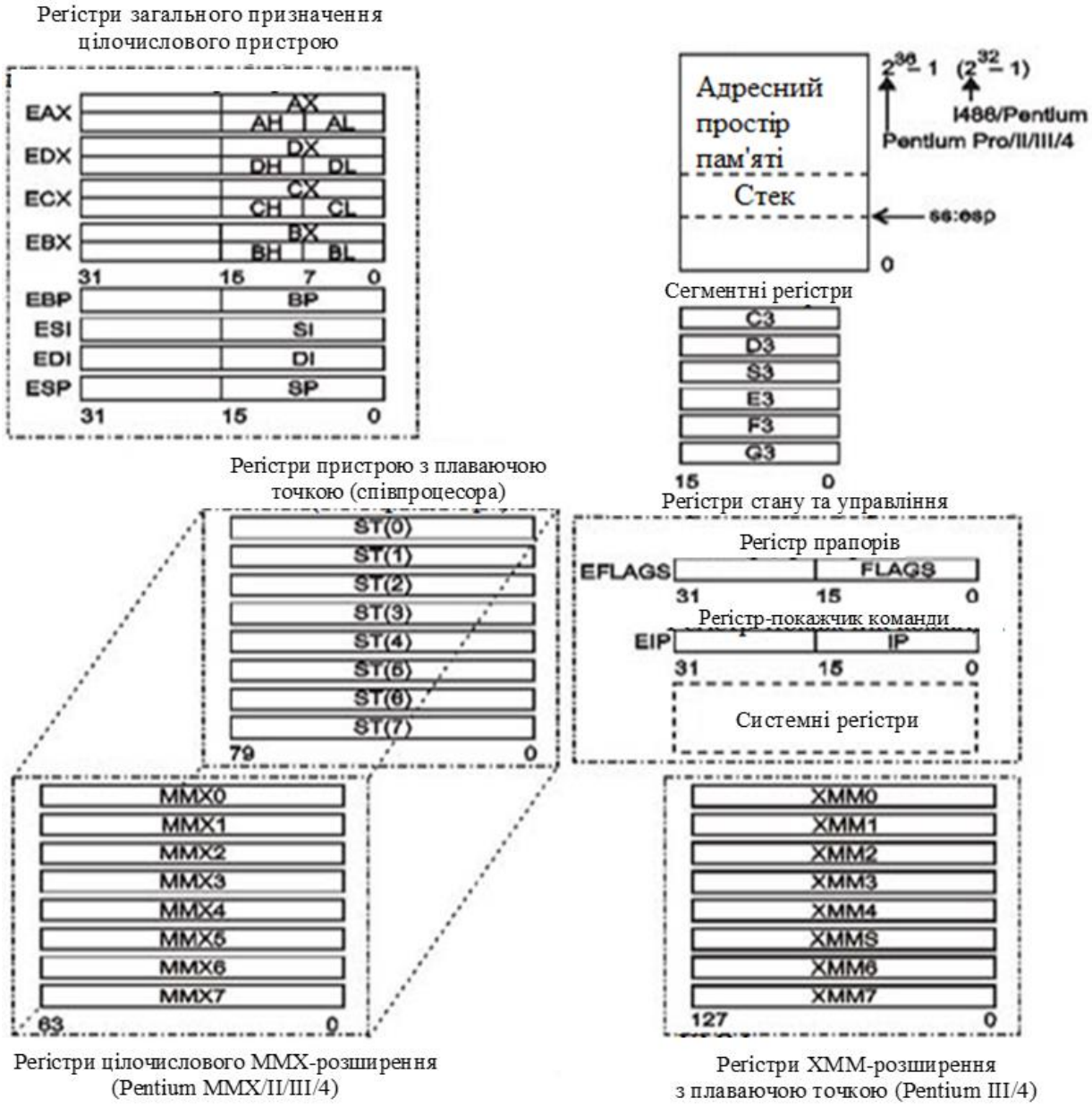


Рисунок А.1 – Програмна модель архітектури IA-32 процесорів Intel

Властивості деяких розглянутих далі програмно-доступних ресурсів визначаються поточним режимом роботи процесора.

Режим роботи процесора визначає поведінку, номенклатуру і властивості доступних ресурсів процесора. Переведення процесора з одного режиму в інший здійснюється спеціальними програмними, а також апаратними методами.

У рамках архітектури IA-32 доступні такі режими роботи процесора.

1. Режим реальних адрес, або просто реальний режим (Real mode) – це режим, в якому працював процесор 18086. Наявність його в i486 і Pentium зумовлено тим, що фірма Intel намагається забезпечити у нових моделях процесорів можливість функціонування програм, розроблених для ранніх моделей.

2. Захищений режим (Protected mode) дозволяє максимально реалізувати всі ідеї, закладені в процесорах архітектури IA-32, починаючи з процесорів i80286. Програми, розроблені для J8086 (реального режиму), не можуть функціонувати в захищеному режимі. Одна з причин цього пов'язана з особливостями формування фізичної адреси в захищеному режимі.

3. Режим віртуального процесора 8086 призначений для організації багатозадачної роботи програм, розроблених для реального режиму (процесора 18086), спільно з програмами захищеного режиму. Перехід у цей режим (virtual 8086 mode) можливий, якщо процесор уже знаходиться у захищеному режимі. Робота програм реального режиму в режимі віртуального процесора i8086 можлива завдяки тому, що процес формування фізичної адреси для них проводиться за правилами реального режиму.

4. Режим системного управління (System Management Mode, SMM) – це новий режим роботи процесора, що вперше з'явився у процесорі Pentium. Він забезпечує операційну систему механізмом для виконання машинозалежних функцій, таких як переведення комп'ютера в режим зниженого енергоспоживання або виконання дій із захисту системи. Для переходу в цей режим процесор повинен отримати спеціальний сигнал SMI від удосконаленого програмованого контролера переривань (Advanced Programmable Interrupt Controller, APIC), при цьому зберігається стан обчислювального середовища процесора. Функціонування процесора у цьому режимі подібне до його роботи у режимі реальних адрес. Вихід із цього режиму здійснюється спеціальною командою процесора. Процесор завжди починає роботу в реальному режимі.

Регістрами називаються області високошвидкісної пам'яті, розташовані всередині процесора в безпосередній близькості від його виконавчого ядра. Доступ до них здійснюється незрівнянно швидше, ніж до комірок оперативної пам'яті. Відповідно машинні команди з операндами у регістрах виконуються максимально швидко, тому в програмах, написаних мовою Assembler, регістри використовуються дуже інтенсивно. З точки зору програміста їх можна розділити на дві великі групи.

Першу групу утворюють призначені для користувача регістри, до яких належать:

– регістри загального призначення EAX/AX/AH/AL, EBX/BX/BH/BL, EDX/DX/DH/DL, ECX/CX/CH/CL, EBP/BP, ESI/SI, EDI/DI, ESP/SP. Вони призначені для зберігання даних і адрес, програміст може їх використовувати (з певними обмеженнями) для реалізації своїх алгоритмів;

– сегментні регістри CS, DS, SS, ES, FS, GS використовуються для зберігання адрес сегментів у пам'яті;

– регістри сопроцесора ST (0), ST (1), ST (2), ST (3), ST (4), ST (5), ST (6), ST (7) призначені для написання програм, що використовують тип даних із плаваючою точкою;

– цілочислові регістри MMX-розширення MMXO, MMXl, MMX2, MMX3, MMX4, MMX5, MMXб, MMX7;

– регістри MMX-розширення з плаваючою точкою XMMO, XMM1, XMM2, XMM3, XMM4, XMM5, XMMб, XMM7;

– регістри стану і управління (регістр прапора E FLAGS/FLAGS і регістр-покажчик команди EIP/IP) містять інформацію про стан процесора або виконуваної програми і дозволяють змінити цей стан.

До другої групи належать системні регістри, тобто регістри, призначені для підтримки різних режимів роботи, сервісних функцій, а також регістри, специфічні для певної моделі процесора. Наведемо перелік системних регістрів, підтримуваних у програмній моделі архітектури IA-32:

– управляючі регістри CRO…CR4 визначають режим роботи процесора і характеристики поточного виконуваного завдання;

– регістри управління пам'яттю GDTR, IDTR, LDTR і TR використовуються у захищеному режимі роботи процесора для локалізації керуючих структур цього режиму;

– налагоджувальні регістри DR0...DR7 призначені для моніторингу та управління різними аспектами налагодження;

– регістри типів областей пам'яті MTRR використовуються для апаратного управління кешуванням для призначення відповідних властивостей областям пам'яті;

– машинозалежні регістри MSR використовуються для управління процесором, контролю за його продуктивністю, отримання інформації про помилки.

У позначеннях багатьох регістрів загального призначення присутня похила розділова риска. Це не різні регістри, а частини одного великого 32-розрядного регістра, але їх можна використовувати у програмі як окремі об'єкти. Процесори i486 і Pentium мають в основному 32-розрядні регістри, вони позначаються приставкою E (Extended).

Багато з наведених регістрів призначені для роботи з певними обчислювальними підсистемами процесора: співпроцесором і MMX-розширеннями.

Регістри загального призначення

Усі регістри цієї групи дозволяють звертатися до своїх молодших частин (рисунок А.2). Для самостійної адресації можна використовувати тільки молодші 16- і 8-бітні частини цих регістрів. Старші 16 біт цих регістрів як самостійні об'єкти недоступні. Це зроблено для сумісності з молодшими 16-розрядними моделями мікропроцесорів фірми Intel.

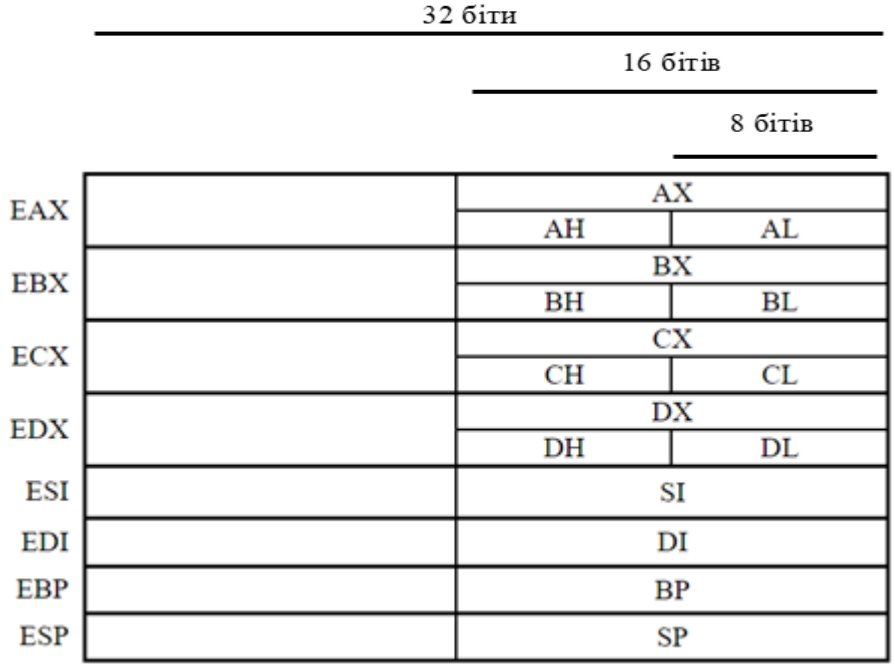


Рисунок А.2 – Реґістри загального призначення

Наведемо перелік реґістрів, які належать до групи реґістрів загального призначення:

– RAX/EAX/AX/AH/AL (Accumulator register) – акумулятор. Застосовується для зберігання проміжних даних. У деяких командах використання цього реґістра обов'язкове;

– RBX/EBX/BX/BH/BL (Base register) – базовий реґістр. Часто застосовується для зберігання базової адреси деякого об'єкта в пам'яті;

– RCX/ECX/CX/CH/CL (Count register) – реґістр-лічильник. Застосовується в командах, які виконують деякі повторювані дії. Його використання найчастіше неявне і приховане в алгоритмі роботи відповідної команди. Наприклад, команда організації циклу LOOP крім передачі керування команді, яка знаходиться за деякою адресою, аналізує і зменшує на одиницю значення реґістра ECX/CX;

– RDX/EDX/DX/DH/DL (Data register) – реґістр даних. Так само, як і реґістр EAX/AX/AH/AL, він зберігає проміжні дані. У деяких командах його використання обов'язкове; для деяких команд це відбувається неявно.

Наступні два реґістри використовуються для підтримки так званих рядкових операцій, тобто операцій, які проводять послідовне оброблення рядків елементів, кожен з яких може мати довжину 32, 16 або 8 бітів:

– RSI/ESI/SI (Source Index register) – індекс джерела. Цей реґістр у ланцюгових операціях містить поточну адресу елемента в рядку-джерелі;

– RDI/EDI/DI (Destination Index register) – індекс приймача (одержувача). Цей реґістр у ланцюгових операціях містить поточну адресу в рядку-приймачі.

В архітектурі мікропроцесора на програмно-апаратному рівні підтримується така структура даних, як *стек*. Стек розташовується в оперативній пам'яті і зазвичай використовується для збереження адреси повернення з підпрограми, для передачі параметрів у підпрограмі і розміщення локальних змінних. Для роботи зі стеком у системі команд мікропроцесора є спеціальні команди, а в програмній моделі мікропроцесора для цього існують спеціальні реґістри:

– RSP/ESP/SP (Stack Pointer register) – реґістр покажчика стека. Містить покажчик вершини стека в поточному сегменті стеку;

– RBP/EBP/BP (Base Pointer register) – реґістр покажчика бази кадра стеку. Призначений для організації довільного доступу до даних усередині стеку. Часто реґістр BP/EBP зберігає адресу початку локальних змінних поточної підпрограми.

Усі реґістри загального призначення (крім ESP) можуть використовуватися при програмуванні для зберігання операндів практично у будь-яких поєднаннях. Але деякі команди використовують фіксовані реґістри для виконання своїх дій. Використання жорсткого закріплення реґістрів для деяких команд дозволяє більш компактно кодувати їх машинне виконання.

Сегментні реґістри

У програмній моделі мікропроцесора є шість сегментних реґістрів: CS, SS, DS, ES, GS, FS (рисунок А.3). Їх існування зумовлено специфікою організації та використання оперативної пам'яті мікропроцесорами Intel. Вона полягає у тому, що мікропроцесор апаратно підтримує структурну організацію програми у вигляді трьох частин, які називаються сегментами. Відповідно така організація пам'яті називається сегментною.

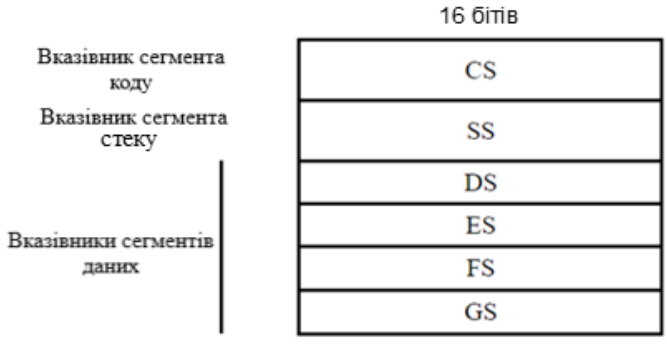


Рисунок А.3 – Сегментні реґістри

Сегментні реґістри призначені для того, щоб указати на сегменти, до яких програма має доступ у конкретний момент часу. Фактично у цих реґістрах містяться адреси пам'яті, з яких починаються відповідні сегменти. Логіка оброблення машинної команди побудована так, що при виконанні

команди доступу до даних програми або до стеку неявно використовуються адреси у певних сегментних реґістрах. Мікропроцесор підтримує такі типи сегментів:

– сегмент коду містить команди програми. Для доступу до цього сегменту служить реґістр CS (code segment register) – сегментний реґістр коду. Він містить адресу сегмента з машинними командами, до якого має доступ процесор (тобто ці команди завантажуються в конвеєр мікропроцесора);

– сегмент даних містить оброблювані програмою дані. Для доступу до цього сегмента служить реґістр DS (data segment register) – сегментний реґістр даних, який зберігає адресу сегмента даних поточної програми;

– сегмент стеку. Цей сегмент є областю пам'яті, що називається стеком. Роботу зі стеком мікропроцесор організує за таким принципом: останній записаний в цю область елемент вибирається першим. Для доступу до цього сегмента служить реґістр SS (stack segment register) – сегментний реґістр стеку, що містить адресу сегмента стеку;

– додатковий сегмент даних. Більшість машинних команд припускають, що оброблювані ними дані розташовані у сегменті даних, адреса якого знаходиться в реґістрі DS.

Якщо програмі недостатньо одного сегмента даних, то вона має можливість використовувати ще три додаткових сегменти даних. Але на відміну від основного сегмента даних, адреса якого міститься у сегментному реґістрі DS, при використанні додаткових сегментів даних їх адреси необхідно вказувати явно за допомогою спеціальних префіксів перевизначення сегментів у команді. Адреси додаткових сегментів даних повинні міститися в реґістрах ES, GS, FS (extension data segment registers).

Реґістри стану і управління

У мікропроцесор включені кілька реґістрів (рисунок А.4), які постійно містять інформацію про стан як самого мікропроцесора, так і команди, які в даний момент завантажені на конвеєр. До цих реґістрів належать:

– реґістр прапорів EFLAGS/FLAGS;

– реґістр покажчика команди EIP/IP.

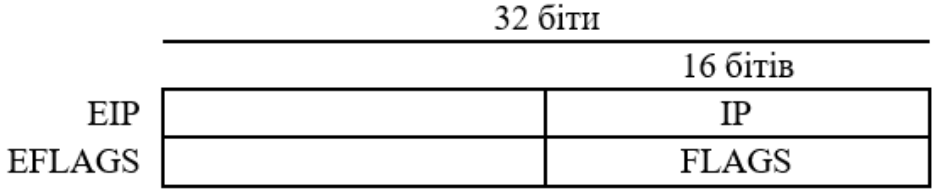


Рисунок А.4 – Реґістри стану і управління

Використовуючи ці реґістри, можна отримувати інформацію про результати виконання команд і впливати на стан самого мікропроцесора. Розглянемо докладніше призначення і вміст цих реґістрів.

EFLAGS/FLAGS (flag register) – реґістр прапорів. Розрядність EFLAGS/FLAGS становить 32/16 бітів. Окремі біти цього реґістра мають певне функціональне призначення і називаються прапорами. Молодша частина цього реґістра повністю аналогічна реґістру FLAGS для мікропроцесора i8086. На рисунку А.5 показано вміст реґістра EFLAGS.

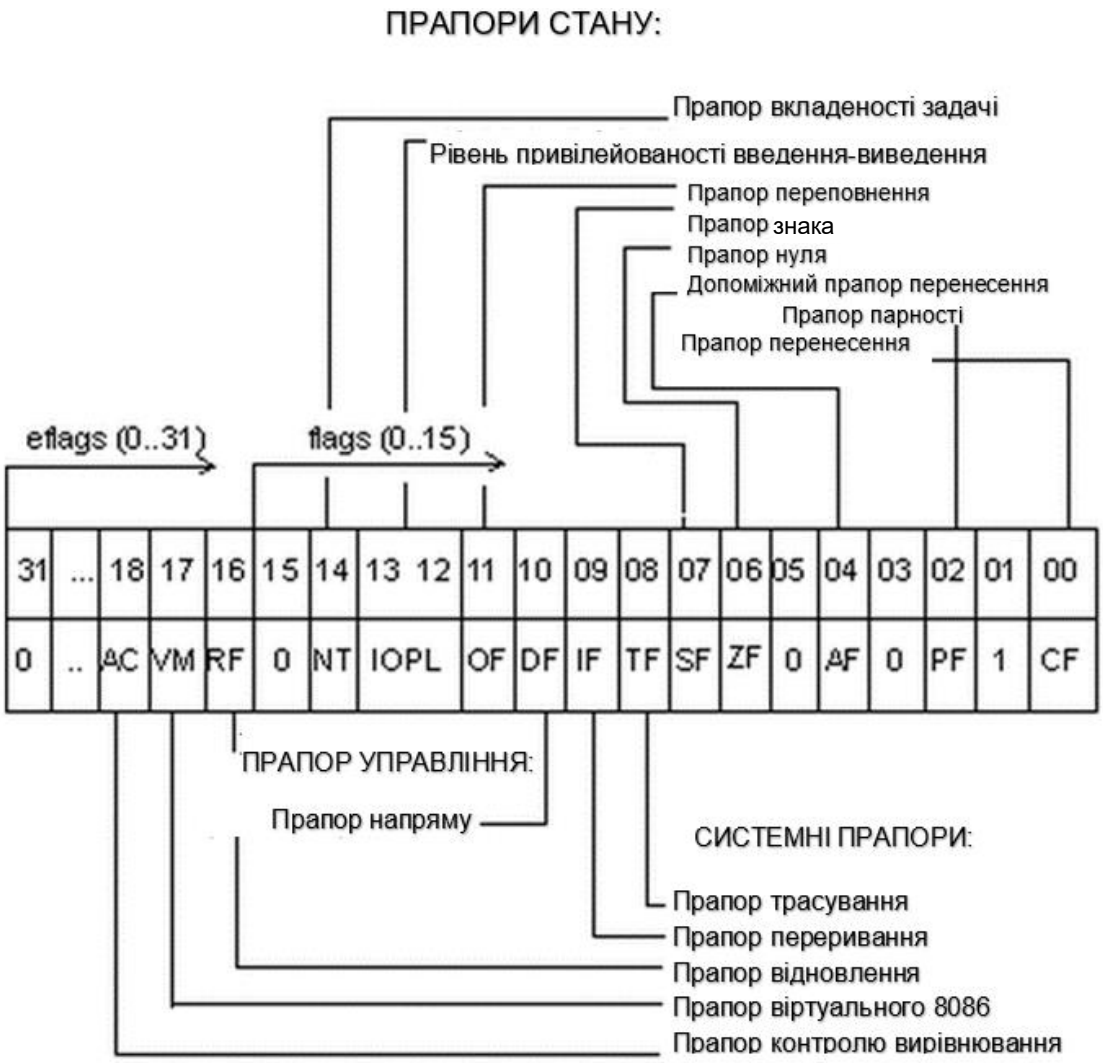


Рисунок А.5 – Уміст реґістра EFLAGS

Виходячи з особливостей використання, прапори реґістра EFLAGS/FLAGS можна розділити на три групи:

1) вісім прапорів стану (таблиця А.1). Ці прапори можуть змінюватися після виконання машинних команд. Прапори стану реґістра EFLAGS відображають особливості результату виконання арифметичних або логічних операцій. Це дає можливість аналізувати стан обчислювального

процесу і реагувати на нього за допомогою умовних команд, наприклад, команд умовних переходів і викликів підпрограм;

2) один прапор напряму. Позначається DF (Directory Flag). Він знаходиться у 10-му біті реґістра EFLAGS і використовується ланцюговими командами. Значення прапора DF визначає напрямок поелементного оброблення в цих операціях: від початку рядка до кінця (DF = 0) або навпаки – від кінця рядка до його початку (DF = 1). Для роботи з прапором DF існують спеціальні команди: CLD (зняти прапор DF) і STD (установити прапор DF). Застосування цих команд дозволяє привести прапор DF у відповідність до алгоритму і забезпечити автоматичне збільшення або зменшення лічильників при виконанні операцій з рядками;

3) п’ять системних прапорів (таблиця А.2), що керують введенням або виведенням, перериванням, налагодженням, перемиканням між завданнями і віртуальним режимом 8086.

Прикладним програмам не рекомендується модифікувати без необхідності ці прапори, оскільки у більшості випадків це призведе до переривання роботи програми.

Таблиця А.1 – Прапори стану

Мнемоніка прапора	Прапор	Номер біта в EFLAGS	Уміст і призначення
CF	Прапор переносу (Carry Flag)	0	1 – арифметична операція зробила перенесення зі старшого біта результату. Старшим є 7-й, 15-й або 31-й біти залежно від розмірності операнда 0 – перенесення не було
PF	Прапор паритету (Parity Flag)	2	1 – вісім молодших розрядів результату містять парну кількість одиниць (цей прапор використовується тільки для восьми молодших розрядів операнда будь-якого розміру) 0 – вісім молодших розрядів результату містять непарну кількість одиниць
ZF	Прапор нуля (Zero Flag)	6	1 – результат нульовий 0 – результат ненульовий

Продовження таблиці А.1

Мнемоніка прапора	Прапор	Номер біта в EFLAGS	Уміст і призначення
AF	Допоміжний прапор перенесення (Auxiliary carry Flag)	4	Тільки для команд, що працюють з BCD-числами. Фіксує факт позичання з молодшої тетради результату: 1 – у результаті операції додавання було проведено перенесення з 3-го розряду у старший розряд або при відніманні було зроблено позику в розряді 3 молодшої тетради із значення в старшій тетраді; 0 – перенесень і позик у (з) 3-й (ого) розряду молодшої тетради результату не було
SF	Прапор знака (Sign Flag)	7	Показує стан старшого біта результату (біти 7-й, 15-й або 31-й для 8-, 16- або 32-розрядних операндів відповідно): 1 – старший біт результату дорівнює 1 0 – старший біт результату дорівнює 0
OF	Прапор переповнення (Overflow Flag)	11	Використовується для фіксування факту втрати значущого біта при арифметичних операціях: 1 – у результаті операції відбулося перенесення (позика) у (зі) старшого, знакового біта результату (біти 7-й, 15-й або 31-й для 8-, 16- або 32-розрядних операндів відповідно) 0 – у результаті операції не відбулося перенесення (позики) у (зі) старшого, знакового біта результату

Продовження таблиці А.1

Мнемоніка прапора	Прапор	Номер біта в EFLAGS	Уміст і призначення
IOPL	Рівень привілеїв введення-виведення (Input/Output Privilege Level)	12, 13	Використовується у захищеному режимі роботи мікропроцесора для контролю доступу до команд введення-виведення залежно від привілейованості задачі
NT	Прапор вкладеності завдання (Nested Task)	14	Використовується у захищеному режимі роботи мікропроцесора для фіксації того факту, що одна задача вкладена в іншу

Таблиця А.2 – Системні прапори

Мнемоніка прапора	Прапор	Номер біта в EFLAGS	Уміст і призначення
TF	Прапор трасування (Trace Flag)	8	Призначений для організації покрокової роботи мікро-процесора: 1 – мікропроцесор генерує переривання з номером 1 після виконання кожної машинної команди. Використовується при налагодженні програм, зокрема налагоджувачами 0 – звичайна робота
IF	Прапор переривання (Interrupt enable Flag)	9	Призначений для дозволу або заборони (маскування) апаратних переривань (переривань по входу INTR): 1 – апаратні переривання дозволені 0 – апаратні переривання заборонені

Продовження таблиці А.2

Мнемоніка прапора	Прапор	Номер біта в EFLAGS	Уміст і призначення
RF	Прапор поновлення (Resume Flag)	16	Використовується при обробленні переривань від реґістрів налагодження
VM	Прапор віртуального режиму роботи (Virtual 8086 Mode)	17	Ознака роботи мікропроцесора у віртуальному режимі 8086: 1 – процесор працює у віртуальному режимі 0 – процесор працює в реальному або захищеному режимі
AC	Прапор контролю вирівнювання (Alignment Check)	18	Призначений для дозволу контролю вирівнювання при зверненнях до пам'яті. Використовується спільно з бітом *am* у системному реґістрі CR0. Наприклад, Pentium дозволяє розміщати команди і дані, починаючи з будь-якої адреси. Якщо потрібно контролювати вирівнювання даних і команд за адресами, кратними 2 або 4, то встановлення цих бітів призведе до того, що всі звернення до адрес, які є некратними, будуть створювати виняткову ситуацію

EIP/IP (Instraction Pointer register) – реґістр-покажчик команд. Реґістр EIP/IP має розрядність 32/16 біт і містить зсув наступної команди, яку необхідно виконати, щодо вмісту сегментного реґістра CS у поточному сегменті команд. Цей реґістр безпосередньо недоступний програмісту, але завантаження і зміна його значення виконуються різними командами управління, до яких належать команди умовних і безумовних переходів, виклику процедур і повернення з процедур. Виникнення переривань також призводить до модифікації реґістра EIP/IP.

Реґістри співпроцесора x87

Співпроцесор (FPU) призначений для виконання операцій над числами. З програмної точки зору співпроцесор містить блок реґістрів даних, реґістр управління і групу реґістрів стану і покажчиків (рисунок А.6).

Вісім реґістрів даних розрядністю 80 бітів організовані в стек. Номер реґістра, що є поточною вершиною стеку, зберігається в спеціальному полі реґістра стану (покажчика вершини стеку). Операція PUSH зменшує значення покажчика на 1 і поміщує дані зі стеку в реґістр, який є новою вершиною стеку. Операція POP записує дані з вершини стеку в пам'ять або реґістр і збільшує покажчик на 1. Інструкції адресують реґістри явно або неявно. Неявна адресація передбачає, що операнд знаходиться на вершині стеку. Явна адресація передбачає указання зсуву реґістра щодо вершини стеку – st (i).

	80 бітів			
Фізичні номери	1 біт Знак	15 бітів Порядок	64 біти Мантиса	Відносні номери
0			mm0	ST(5)
1			mm1	ST(6)
2			mm2	ST(7)
3			mm3	ST(0)
4			mm4	ST(1)
5			mm5	ST(2)
6			mm6	ST(3)
7			mm7	ST(4)

Рисунок А.6 – Реґістри даних FPU (арифметичний стек)

Розширення MMX

MMX було першим розширенням, яке реалізує технологію SIMD (Single Instruction – Multiple Data). Основна ідея SIMD полягає в одночасному обробленні декількох елементів даних однією операцією. Розширення MMX використовує нові типи упакованих 64-бітних цілочислових даних:

– вісім упакованих байтів (Packed byte);

– чотири упакованих слова (Packed word);

– два упакованих подвійних слова (Packed double word);

– одне почетверне слово (Quad word).

Ці типи даних можуть спеціальним чином оброблятися у 64-бітних реґістрах MM0 – MM7, що є молодшими бітами стеку 80-бітних реґістрів FPU. Кожна інструкція MMX виконує дію відразу над усім комплектом операндів (8, 4, 2 або 1), розміщених у реґістрах, що адресуються. Як і реґістри FPU, ці реґістри не можуть використовуватися для адресації пам'яті. Збіг реґістрів MMX і FPU накладає обмеження на чергування кодів

FPU і MMX. На відміну від стеку FPU регістри MMX адресуються не за допомогою стеку, а фізично (за своїми фізичними номерами).

Блок XMM

Починаючи з Pentium III, Intel використовує в своїх процесорах нове потокове розширення SSE (Streaming SIMD Extension). Воно реалізується додатковим незалежним блоком, що має вісім 128-бітних регістрів XMM0–XMM7 і регістр стану та управління MXCSR. У кожному з регістрів XMM розміщуються чотири числа в форматі з плаваючою точкою одинарної точності. Блок дозволяє виконувати векторні (пакетні) і скалярні інструкції. Векторні інструкції реалізують операції відразу над чотирма комплектами операндів. Скалярні інструкції працюють тільки з одним комплектом операндов – молодшим 32-бітним словом. При виконанні інструкцій XMM традиційне обладнання блоків FPU/MMX не використовується, що дозволяє ефективно змішувати інструкції MMX з інструкціями з плаваючою точкою.

Крім інструкцій з новим блоком XMM у розширення SSE входять додаткові цілочислові інструкції з регістрами MMX, а також інструкції управління кешуванням.

У процесорі Pentium4 набір інструкцій отримав нове розширення – SSE2, в основному це стосується додавання нових 128-бітних типів даних для блока XMM:

– упакована пара дійсних чисел подвійної точності;

– упаковані цілі числа: 16 байт, 8 слів, 4 подвійних слова або пара четверних слів.

У процесор уведені нові функції цілочислової арифметики, 128-розрядні для регістрів XMM і такі ж 64-розрядні для регістрів MMX; ряд старих інструкцій MMX поширений на XMM (у 128-бітному варіанті); додані інструкції перетворення для нових форматів даних, а також розширені можливості "перемішування" даних у блоці XMM. Крім того, розширено підтримку управління кешуванням і порядком виконання операцій з пам'яттю.

Регістри архітектури x86-64 (64-бітна архітектура)

Існують такі варіанти назв цієї 64-бітної версії x86:

– x86-64 – початковий варіант. Саме під цією назвою фірмою AMD була опублікована перша специфікація;

– Intel 64 – поточна офіційна назва реалізації архітектури від Intel. Поступово Intel відмовляється від назв «IA-32», «IA-32e» і «EM64T» на користь цього найменування, яке нині є єдиним офіційним для реалізації цієї архітектури від Intel.

Нині найбільш поширені найменування 64-бітної версії x86 – це «x64», «x86-64» і «AMD64».

Представники Intel 64:

– 64-бітні моделі сімейств Pentium 4, Celeron D і Intel Atom;

– сімейства Core 2, Core i3, Core i5, Core i7, Core i9, Xeon.

Представники IA-64: сімейства Itanium і Itanium 2.

Процесори цієї архітектури підтримують два режими роботи (рисунок А.7): Long mode («довгий» режим) і Legacy mode («успадкований» – режим сумісності з 32-бітною архітектурою x86).

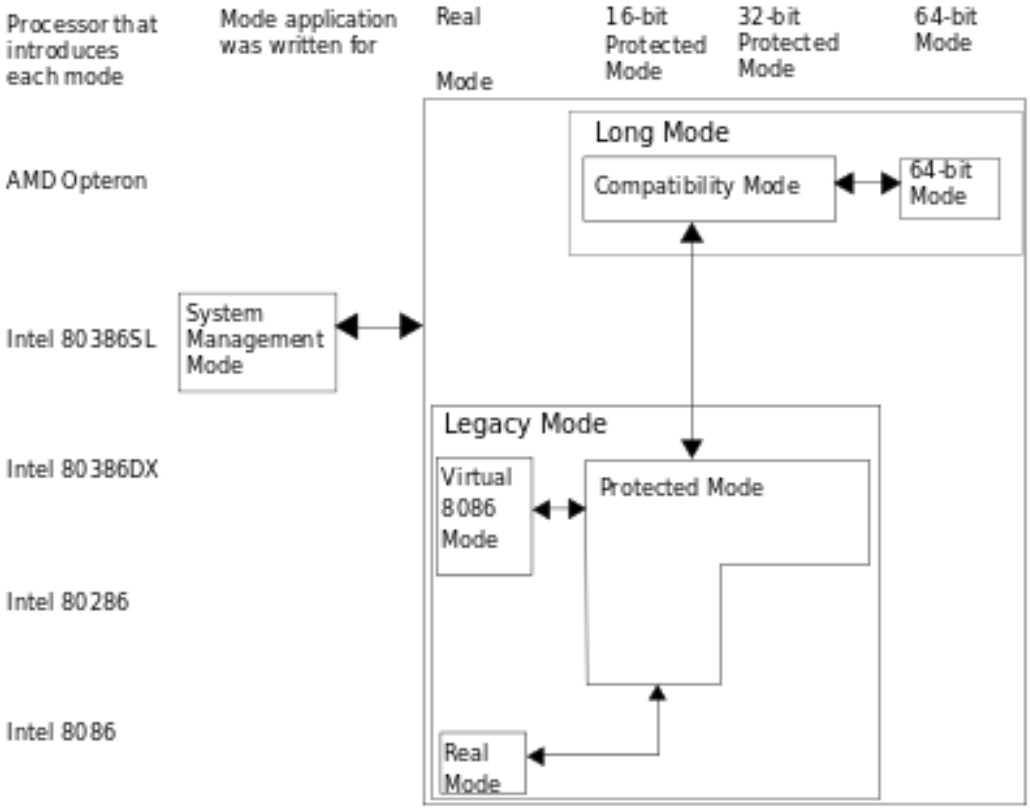


Рисунок А.7 – Режими роботи мікропроцесорів

Long Mode, або «довгий» режим – «рідний» для процесорів AMD64. Цей режим дає можливість скористатися всіма перевагами архітектури x86-64. Для використання цього режиму необхідна будь-яка 64-бітна операційна система (наприклад, Windows Server 2003/2003R2/2008/2008R2/2012, Windows XP Professional x64 Edition, Windows Vista x64, Windows 7/8/8.1/10 x64 або 64-бітові варіанти UNIX-подібних систем GNU/Linux, FreeBSD, OpenBSD, NetBSD (чисті 64-бітні зборки однак мають можливість запуску 32-бітних додатків), Solaris (змішана 32/64 зборка з різними ядрами для 32- і 64-бітних процесорів), Mac OS X (змішана 32/64 зборка з 32-бітним ядром, починаючи з версії 10.4.7).

Цей режим дозволяє виконувати 64-бітні програми. Також (для забезпечення сумісності) надається підтримка виконання 32-бітного коду, наприклад, 32-бітних програмних застосунків (32-бітні програми, навіть якщо вони запущені у 64-бітній системі, не зможуть використовувати 64-бітні системні бібліотеки, і навпаки). Щоб вирішити цю проблему, більшість 64-розрядних операційних систем надає два набори необхідних системних API: один – для «рідних» 64-бітних програмних застосунків, інший – для 32-бітних програмних застосунків (ця методика використовується у ранніх 32-бітних системах, наприклад, Windows 95 і Windows NT, для виконання 16-бітних програм).

У «довгому» режимі скасовано ряд «рудиментів» архітектури x86-32, зокрема таких, як віртуальний режим 8086, сегментна модель пам'яті (проте, залишилася можливість використання сегментів FS і GS, що корисно для швидкого знаходження важливих даних потоку при перемиканні задач), апаратна багатозадачність і ряд команд, що реалізують скасовані можливості, а також робота з BCD-числами, які в нових програмах практично не використовувалися. «Довгий» режим активується установленням прапора CR0.PG, який використовується для включення сторінкового MMU (за умови що таке перемикання дозволено (EFER.LME = 1), в іншому випадку просто відбудеться включення MMU в «успадкованому» режимі). Таким чином, неможливе виконання 64-бітного коду із забороненим сторінковим перетворенням. Це створює певні труднощі в програмуванні, оскільки при перемиканні з «довгого» в «успадкований» режим і назад (наприклад, для виклику функцій BIOS або DOS монітором віртуальної машини тощо) потрібне подвійне скидання MMU, для чого код перемикання повинен знаходитися в тотожно відображеній сторінці.

«Успадкований» режим Legacy Mode дозволяє x86-64-процесору виконувати команди для процесорів x86, і таким чином реалізує повну сумісність з 32-бітним кодом і 32-бітними операційними системами для x86. У цьому режимі процесор поводиться так само, як x86-процесор (наприклад, як Athlon або Pentium III). Функції і можливості, що надаються архітектурою x86-64 (наприклад, 64-бітні регістри), у цьому режимі недоступні. У цьому режимі 64-бітні програми та операційні системи працювати не будуть.

Розроблений компанією AMD набір інструкцій x86-64 (пізніше перейменований в AMD64) є розширенням архітектури Intel IA-32 (X86-32). Основною відмінною рисою AMD64 є підтримка 64-бітових регістрів загального призначення, 64-бітових арифметичних і логічних операцій над цілими числами і 64-бітних віртуальних адрес. Для адресації нових регістрів для команд уведені так звані «префікси розширення регістру», для яких був вибраний діапазон кодів 40h-4Fh, що використовуються для

команд INC <реґістр> і DEC <реґістр> у 32-бітних режимах. Команди INC і DEC у 64-бітному режимі повинні кодуватися у більш загальній двобайтовій формі.

Архітектура x86-64 має (рисунок А.8):

– 16 цілочислових 64-бітних реґістрів загального призначення (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 – R15);

– вісім 80-бітних реґістрів з плаваючою точкою (ST0 – ST7);

– вісім 64-бітних реґістрів Multimedia Extensions (MM0 – MM7, мають спільний простір з реґістрами ST0 – ST7);

– 16 128-бітних реґістрів SSE (XMM0 – XMM15);

– 64-бітний покажчик RIP і 64-бітний реґістр прапорів RFLAGS.

Також подвоюється кількість реґістрів XMM (рисунок А.9). Реґістри FPU/MMX залишаються без змін.

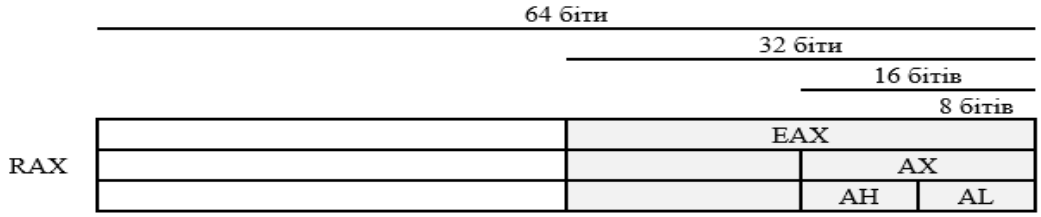


Рисунок А.8 – Архітектура x86-64

На відміну від процесорів x86, в яких усі обчислення з плаваючою точкою проводилися у співпроцесорі, а блоку XMM відводилися тільки векторні операції, процесори x86-64 практично всі обчислення з плаваючою точкою виконують у блоці XMM.

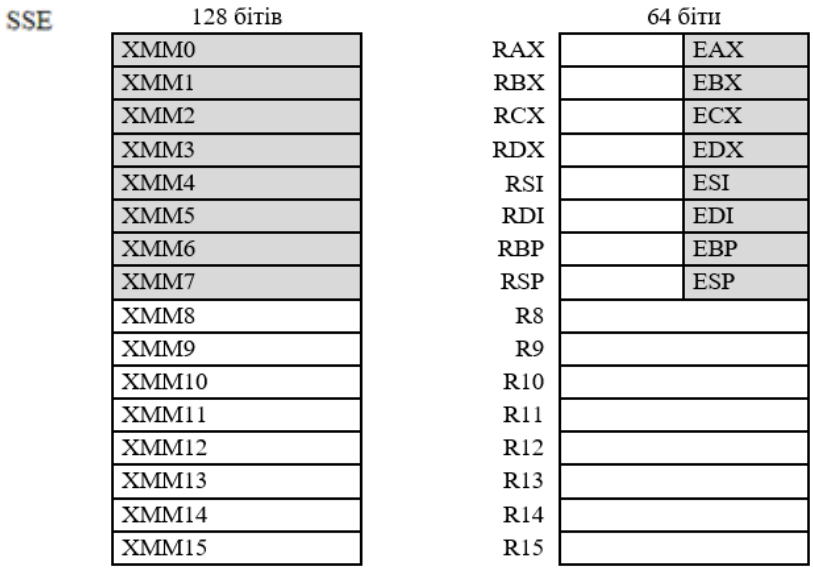


Рисунок А.9 – Реґістри XMM-розширення

ДОДАТОК Б

Створення застосунків мовою ASSEMBLER. Процедури та їх виклик

Процес проєктування

Пакет MASM містить основні засоби, необхідні для виконання всіх кроків розроблення програм. Він підтримує такі етапи створення та виконання програм:

1. Асемблювання вихідного тексту програми (файл з розширенням *asm*), у результаті якого формується об'єктний файл (з розширенням *obj*). Асемблювання виконує утиліта **ml.exe**, виклик якої з командного рядка може мати такий вигляд:

ml/c/coff імя_файла.asm

Транслятор **ml.exe** створює об'єктний файл, який в цьому випадку має формат **coff**, застосовуваний при компонуванні із застосунками, розробленими в середовищі Visual Studio.

2. Об'єднання (компонування, зборка) отриманого об'єктного файла з іншими об'єктними файлами і бібліотеками у виконуваний файл (з розширенням *exe*). Для побудови 32-розрядних застосунків можна використовувати команду

Link/Subsystem: windows/opt: noref імя_файла.obj

або

Link/Subsystem: console/opt: noref імя_файла.obj.

3. Виконання отриманого *exe*-файла проводиться звичайним способом

ім’я_файла.exe список параметрів.

Структура програми

Програми, написані мовою Assembler, зазвичай містять один або декілька сегментів. Сегментом називають частину програми, що складається з команд і/або даних.

Кожний сегмент має певне функціональне призначення, наприклад, сегмент для зберігання даних, сегмент команд програми, сегмент стека. Для процесорів Intel сегменти розміщуються в оперативній пам'яті окремо і незалежно один від одного. Для вказівки початкової (базової) адреси сегмента призначені спеціальні сегментні реґістри CS, DS, SS, ES, FS, GS. Кожен з них використовується для сегментів певного призначення:

– CS (Code Segment) – для зберігання базової адреси сегмента команд;

– SS (Stack Segment) – для сегмента стеку;

– DS (Data Segment), ES (Extra Segment), FS, GS – для сегментів даних.

Програма, крім команд процесора, містить спеціальні інструкції, які вказують самому Assembler на виконання певних дій, наприклад, на визначення змінних, сегментів.

Для визначення сегмента використовуються директиви **SEGMENT** і **ENDS**:

ім’я_сегмента SEGMENT [Параметри]
...
ім’я_сегмента ENDS.

Програма мовою Assembler може мати такий вигляд:

```
; визначення сегмента1
; ...
; визначення сегмента n END [точка входу].
```

Директива **END** завершує текст asm-файла, точка входу задається міткою першої виконуваної команди.

Директива **ASSUME** вказує, з яким сегментом або групою сегментів пов'язаний той чи інший сегментний реґістр. Зазвичай вона використовується так:

ASSUME CS: ім’я_сегмента_коду, DS: ім’я_сегмента_даних, SS: ім’я_сегмента_стеку.

За замовчуванням використовуються команди процесора 8086. Для використання команд більш сучасних процесорів використовуються відповідні директиви, наприклад:

.**386**.

Для визначення даних у сегменті використовуються директиви **DB**, **DW**, **DD** та інші.

Приклади використання директив:

```
a     dw    13
b     db    'Hello', 13,10, '$';        рядок з восьми байтів
з     dd    ?;                          неініціалізована змінна
ar1   dd    10,20,30,40,50;             масив з п'яти подвійних слів
ar2   dw    10 dup (?);          масив з десяти неініціалізованих слів
ar3   db    15 dup (0);                 масив з 15 нулів
```

Моделі пам'яті

Для спрощення процесу розроблення застосунків використовують моделі пам'яті. Модель визначає спосіб організації адресного простору оперативної пам'яті: кількість сегментів і правила розміщення. Визначити модель можна директивою

.MODEL ім’я_моделі.

Різні моделі пам'яті можуть використовувати один або кілька сегментів коду і даних. Для доступу до їх вмісту вказуються адреси типу FAR (селектор_сегмент: зсув) і NEAR (зсув).

Приклади моделей та їх характеристики наведено в таблиці Б.1.

Таблиця Б.1 – Моделі пам’яті

Модель	Тип адресації коду	Тип адресації даних	Призначення моделі
TINY	near	near	Для створення невеликих програм. Код, дані і стек розміщуються в одному сегменті розміром до 64 Кб
SMALL	near	near	Цю модель використовують для більшості програм на Асемблері. Код займає один сегмент, дані і стек – інший.
MEDIUM	far	near	Код займає кілька сегментів, усі дані – один
COMPACT	near	far	Код – в одному сегменті; дані – у кількох
LARGE	far	far	Код і дані можуть займати кілька сегментів
FLAT	near	near	Для розроблення 32-бітних застосунків. Код, дані і стек розміщуються в одному сегменті розміром до 4 Гб.

Спрощені директиви сегментації

Макроасемблер MASM містить директиви, що спрощують визначення сегментів програми і підтримують угоди з управління сегментами в мовах високого рівня (таблиця Б.2).

Таблиця Б.2 – Спрощені директиви визначення сегмента

Директива	Призначення
.CODE	Початок або продовження сегмента коду
.DATA	Початок або продовження сегмента ініціалізованих даних. Також використовується для визначення даних типу near
.CONST	Початок або продовження сегмента постійних даних (констант) модуля
.DATA?	Початок або продовження сегмента неініціалізованих даних. Також використовується для визначення даних типу near
.STACK [розмір]	Початок або продовження сегмента стеку модуля. Параметр [розмір] задає розмір стеку
.FARDATA	Початок або продовження сегмента ініціалізованих даних типу far
.FARDATA?	Початок або продовження сегмента неініціалізованих даних типу far

Модель пам'яті FLAT

У «плоскій» моделі управління пам'яттю flat всі адреси є near-адресами і визначаються 32-бітними зсувами від базової адреси відповідного сегмента. Ця модель найчастіше використовується компіляторами мов високого рівня.

Процедури

Часто програмі потрібно тимчасово запам'ятати інформацію. Для цього в програмі використовується спеціальний сегмент – сегмент стеку, який називають просто *стеком*. При переміщенні елементів у стек відбувається зменшення покажчика вершини стеку, а при вилученні – збільшення. Тобто стек завжди «зростає» у бік менших адрес пам'яті.

Для роботи зі стеком використовуються реґістри SS, ESP і EBP. Уміст SS є базою стеку. У ESP зберігається зсув вершини стеку. Спочатку ESP ініціалізується найбільшим зсувом, якого може досягати стек, потім змінюється операціями включення і вилучення. Реґістр EBP зазвичай використовується для звернень до елементів стеку.

Нижче розглянуті деякі команди роботи зі стеком.

Команда збереження даних у стек:

PUSH джерело.

Джерелом можуть бути реґістр, сегментний реґістр або змінна. ESP зменшується на розмір джерела в байтах (2 або 4) і вміст джерела поміщується в пам'ять за адресою SS: ESP.

Команда вилучення даних зі стеку

POP приймач

поміщує в приймач слово або подвійне слово, яке знаходиться у вершині стеку, збільшуючи ESP на 2 або 4 відповідно. Приймачем може бути реґістр загального призначення, сегментний реґістр (крім CS) або змінна. Якщо приймачем є операнд, який використовує ESP для непрямої адресації, команда POP обчислює адресу операнда вже після того, як вона збільшує ESP.

Команда **PUSH** часто використовується в парі з **POP**.

Копіювання одного сегментного реґістра в інший (що можна виконати однією командою **MOV**), реалізується так:

```
PUSH  DS
POP   ES
```

Щоб зберегти/витягти в/зі стеку реґістр прапорів, використовуються команди **PUSHF/POPF**.

БІБЛІОГРАФІЧНИЙ СПИСОК

Галисеев, Г. В. Ассемблер IBM PC. Самоучитель / Г. В. Галисеев. – М. : Изд. дом «Вильямс», 2004. – 304 с.

Зубков, С. В. Ассемблер для DOS, Windows и UNIX / С. В. Зубков. – М. : ДМК Пресс, 2000. – 608 с.

Лямин, Л. В. Макроассемблер MASM / Л. В. Лямин. – М. : Радио и связь, 1994. – 320 с.

Магда, Ю. Ассемблер для процессоров Intel Pentium / Ю. Магда. – Спб. : Питер, 2006. – 410 с.

Юров, В. Assembler : спец. справ. / В. Юров. – СПб. : Питер, 2001. – 496 с.

Юров, В. Assembler : учеб. для вузов / В. Юров. – 2-е изд. – СПб. : Питер, 2007. – 637 с.

ЗМІСТ

Навчальне видання

Меняйлов Євген Сергійович,
Базілевич Ксенія Олексіївна,
Трофимова Ірина Олексіївна та ін.

СПІЛЬНЕ ВИКОРИСТАННЯ МОВИ ASSEMBLER ТА МОВ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ

Редактор С. П. Гевло

Зв. план, 2021
Підписано до видання 24.12.2021
Ум. друк. арк. 6,4. Обл.-вид. арк. 7,25. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001