Міністерство освіти і науки України

Полтавський національний технічний університет
імені Юрія Кондратюка

Кафедра комп’ютерних та інформаційних технологій і систем

НАВЧАЛЬНИЙ ПОСІБНИК З ДИСЦИПЛІНИ **«КОМП'ЮТЕРНА СХЕМОТЕХНІКА ТА АРХІТЕКТУРА КОМП'ЮТЕРІВ»** ДЛЯ СТУДЕНТІВ 2 КУРСУ СПЕЦІАЛЬНОСТІ 122 «КОМП’ЮТЕРНІ НАУКИ»

Полтава 2017

Навчальний посібник з дисципліни «Комп'ютерна схемотехніка та архітектура комп'ютерів» для студентів 2 курсу спеціальності 122 «Комп’ютерні науки». – Полтава: ПолтНТУ, 2017. – 98 с.

Укладачі: М. І. Демиденко ст. викладач, О. А.Руденко к.т.н., доцент кафедри

Відповідальний за випуск: О.Л. Ляхов, завідувач кафедри комп’ютерних інформаційних технологій і систем, доктор технічних наук, професор

Рецензент:, к.т.н. Головко Г.В., доцент

Затверджено науково-методичною радою університету
від _____20_ р.,
протокол № __

Авторська редакція

40.34.6.3

ЗМІСТ

ВСТУП

Навчальний посібник з дисципліни «Комп'ютерна схемотехніка та архітектура комп'ютерів» складена відповідно до місця та значення дисципліни за структурно-логічною схемою, передбаченою освітньо-професійною програмою підготовки бакалаврів спеціальності 122 «Комп'ютерні науки».

Мета дисципліни: забезпечити отримання студентами теоретичних знань з архітектури обчислювальних систем і практичних навичок програмування низького рівня.

Предмет дисципліни: архітектура сучасних ЕОМ.

Структурно-логічне місце дисципліни: попереднє вивчення дисциплін «Алгоритмізація та програмування», «Вища математика».

Завдання дисципліни: В результаті вивчення дисципліни студенти повинні

знати:

– системи числення та кодування в них числової інформації в прямому та додатковому коді;

– архітектуру мікропроцесорів (МП) сімейства Intel;

– послідовність роботи вузлів МП;

– основні команди МП Intel;

– види адресації та їх реалізації;

– шини інтерфейсів, процесорів та периферійних пристроїв;

– організацію системних обмінів інформацією між вузлами МП, між МП та периферією;

– організацію системних переривань;

– архітектуру сучасних ЕОМ.

вміти:

– розробляти специфікації комп'ютерного обладнання, засобів зв'язку та обслуговування (2.ПФ.Е.03.06);

– тестувати й налагоджувати апаратно-програмні засоби і комплекси систем автоматизації та управління (4.ПФ.С.01.03).

ЧАСТИНА І КОМП'ЮТЕРНА СХЕМОТЕХНІКА

1.1 Логічні елементи

Логічний елемент – пристрій, призначений для обробки інформації в цифровій формі (послідовності сигналів високого – «1» і низького – «0» рівнів у двійковій логіці, послідовність «0», «1» та «2» в трійковій логіці, послідовності «0», «1», «2», «3», «4», «5», «6», «7», «8» та «9» в десятковій логіці). Фізично логічні елементи можуть бути виконані механічними, електромеханічними (на електромагнітних реле), електронними (на діодах і транзисторах), пневматичними, гідравлічними, оптичними та іншими способами.

Логічні елементи виконують логічну функцію (операцію) над вхідними сигналами (операндами, даними).

Функція $y=f(x_1, x_2,..., x_n)$ називається перемикальною, або логічною, якщо сама функція y і кожен з її аргументів x_i, приймають значення тільки із множини $\{0,1\}$.

Всього можливо $x^{(x^n)*m}$ логічних функцій і відповідних їм логічних елементів, де x – основа системи числення, n – число входів (аргументів), m – число виходів, тобто нескінченне число логічних елементів. Тому в даному посібнику розглядаються тільки найпростіші і найважливіші логічні елементи.

Всього можливо 16 двійкових двовхідних логічних елементів та 256 двійкових тривхідних логічних елементів (булева функція).

Двійкові логічні операції з цифровими сигналами (бітові операції).

Логічні операції (булева функція) своє теоретичне обґрунтування отримали в математичній логіці.

Логічні операції з одним операндом називаються унарними, з двома – бінарними, з трьома – тернарними і т. д.

Із чотирьох можливих унарних операцій з унарним виходом інтерес для реалізації представляють операції заперечення і повторення, причому, операція заперечення має більше значення, ніж операція повторення, оскільки повторювач може бути зібраний з двох інверторів, а інвертор з повторювачів не зібрати.

Заперечення. Операція "НЕ"

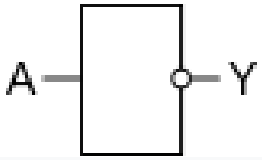


Рисунок 1.1 – Інвертор (IEC)

Рисунок 1.2 – Інвертор (ANSI)

0	1
1	0

Мнемонічне правило для «НЕ» звучить так:
На виході буде:
– «1» тоді і лише тоді, коли на вході «0»,
– «0» тоді і лише тоді, коли на вході «1»

Повторення

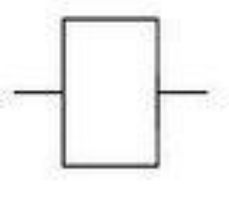


Рисунок 1.3 – Повторювач

0	0
1	1

Перетворення інформації вимагає виконання операцій з групами знаків, найпростішою з яких є група з двох знаків. Оперування з великими групами завжди можна розбити на послідовні операції з двома знаками.

Із можливих бінарних логічних операцій з двома знаками та унарним виходом інтерес для реалізації представляють 10 операцій, наведених нижче.

Кон'юнкція. Операція «І»

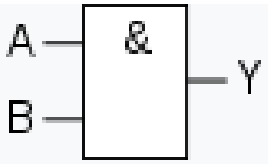


Рисунок 1.4 – Логічний елемент «І» (IEC)

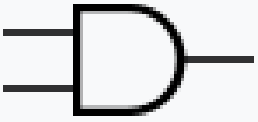


Рисунок 1.5 – Логічний елемент «І» (ANSI)

0	0	0
1	0	0
0	1	0
1	1	1

Логічний елемент, що реалізує функцію кон’юнкції, називається схемою збігу. Мнемонічне правило для І з будь-якою кількістю входів звучить так: на виході буде:

– «1» тоді і тільки тоді, коли на ***всіх*** входах діють «1»,

– «0» тоді і тільки тоді, коли ***хоча б на одному*** вході діє «0»

Словесно цю операцію можна виразити таким виразом: «Істина на виході може бути при істині на вході 1 та істині на вході 2».

Диз'юнкція. Операція «АБО»

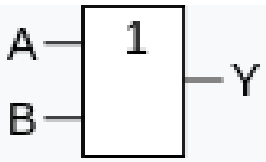


Рисунок 1.6 – Логічний елемент «АБО» (IEC)

Рисунок 1.7 – Логічний елемент «АБО» (ANSI)

0	0	0
1	0	1
0	1	1
1	1	1

Мнемонічне правило для АБО з будь-якою кількістю входів звучить так: На виході буде:

– «1» тоді і тільки тоді, коли ***хоча б на одному*** вході діє «1»,

– «0» тоді і тільки тоді, коли на ***всіх*** входах діють «0».

Заперечення кон'юнкції. Операція "І-НЕ" (штрих Шефера)

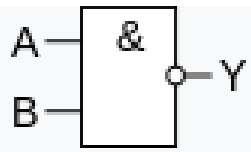


Рисунок 1.8 – Логічний елемент «І-НЕ» (IEC)

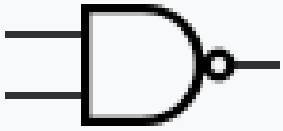


Рисунок 1.9 – Логічний елемент «І-НЕ» (ANSI)

0	0	1
0	1	1
1	0	1
1	1	0

Мнемонічне правило для І-НЕ з будь-якою кількістю входів звучить так: На виході буде:

– «1» тоді і тільки тоді, коли ***хоча б на одному*** вході діє «0»,

– «0» тоді і тільки тоді, коли на ***всіх*** входах діють «1».

Заперечення диз'юнкції. Операція "АБО-НЕ" (стрілка Пірса)

В англомовній літературі NOR.

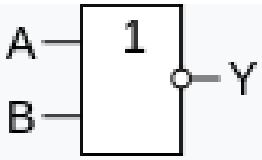


Рисунок 1.10 – Логічний елемент «АБО-НЕ» (IEC)

Рисунок 1.11 – Логічний елемент «АБО-НЕ» (ANSI)

		↓
0	0	1
0	1	0
1	0	0
1	1	0

Мнемонічне правило для АБО-НЕ з будь-якою кількістю входів звучить так: На виході буде:

– «1» тоді і тільки тоді, коли на ***всіх*** входах діють «0»,

– «0» тоді і тільки тоді, коли ***хоча б на одному*** вході діє «1».

Еквіваленція. Операція «ВИКЛЮЧНЕ_АБО-НЕ»

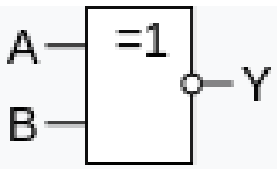


Рисунок 1.12 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО-НЕ» (IEC)

Рисунок 1.13 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО-НЕ» (ANSI)

		↔
0	0	1
0	1	0
1	0	0
1	1	1

Мнемонічне правило ВИКЛЮЧНЕ_АБО-НЕ з будь-якою кількістю входів звучить так: На виході буде:

– «1» тоді і тільки тоді, коли на вході діє ***парна*** кількість,

– «0» тоді і тільки тоді, коли на вході діє ***непарна*** кількість.

Словесний опис: "істина на виході при істині на вході 1 **і** вході 2 **або** при хибності на вході 1 **і** вході 2".

Виключна диз'юнкція. Операція ВИКЛЮЧЕНЕ_АБО

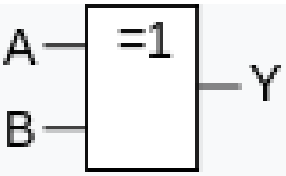


Рисунок 1.14 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО» (IEC)

Рисунок 1.15 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО» (ANSI)

В англомовній літературі XOR (від англ. *exclusive OR*).

0	0	0
0	1	1
1	0	1
1	1	0

Мнемонічне правило для ВИКЛЮЧНЕ_АБО з будь-якою кількістю входів звучить так: На виході буде:

– «1» тоді і тільки тоді, коли на вході діє ***непарна*** кількість,

– «0» тоді і тільки тоді, коли на вході діє ***парна*** кількість.

Словесний опис: «істина на виході – **тільки** при істині на вході 1, або **тільки** при істині на вході 2» [1].

1.2 Тригер

Тригер (англ. trigger, flip-flop) – електронна логічна схема, яка має два стійкі стани, в яких може перебувати, доки не зміняться відповідним чином сигнали керування. Напруги і струми на виході тригера можуть змінюватися стрибкоподібно.

В арифметичних і логічних пристроях для збереження інформації найчастіше використовують тригери – пристрої з двома стійкими станами по виходу, які містять елементарну запам’ятовувальну комірку бістабільна схема (БС) і схему керування (СК). Схема керування перетворює інформацію, яка надходить, на комбінацію сигналів, що діють безпосередньо на входи елементарної запам’ятовувальної комірки. Для забезпечення надійного перемикання в точках А для деяких тригерів повинні бути кола затримки. З цією метою можуть використовуватися запам’ятовувальні елементи на основі БС того ж типу, що вже є у тригері. Схему такого тригера називають схемою типу M-S (master-slave), оскільки стан однієї БС, яку називають веденою, повторює стан додаткової БС, яку називають ведучою.

Тригери широко використовуються для формування імпульсів, у генераторах одиничних сигналів, для побудови подільників частоти,

лічильників, перерахункових пристроїв, регістрів, суматорів, у пристроях керування тощо.

У більшості серій інтегральних елементів містяться тригери різних типів, у тому числі універсальні.

Класифікація тригерів:

– за способом організації логічних зв'язків розрізняють тригери з запуском (RS-тригери), з лічильним входом (Т-тригери), тригери затримки (D-тригери), універсальні (JK-тригери), комбіновані (наприклад, RST-, JKRS-, DRS-тригери);

– за способом запису інформації тригери поділяють на несинхронізовані (асинхронні, нетактові) і синхронізовані (тактові);

– за кількістю інформаційних входів тригери можуть бути з одним, двома та багатьма входами;

– за видом вихідних сигналів тригери поділяються на статичні і динамічні. Статичні тригери – тригери, в яких вихідні сигнали в стійких станах залишаються незмінними в часі. Динамічні тригери – тригери, в яких вихідні сигнали в стійких станах змінюються в часі;

– за способом запам'ятовування інформації тригери можуть бути з логічною і фізичною організацією пам'яті. Перші виконують на логічних елементах І, АБО, НІ, І-НІ, АБО-НІ, І-АБО-НІ і т.д., а другі є елементами запам'ятовувальних пристроїв, у яких використовують нелінійні властивості матеріалів або нелінійні вольт-амперні характеристики компонентів.

RS-тригери

RS-тригер, або SR-тригер – тригер, який зберігає свій попередній стан при нульових входах та змінює свій вихідний стан при подачі на один з його входів одиниці.

При подачі одиниці на вхід S (від англ. Set – встановити) вихідний стан стає рівним логічної одиниці. А при подачі одиниці на вхід R (від англ. Reset – скинути) вихідний стан стає рівним логічному нулю. Стан, при якому на обидва входи R і S одночасно подані логічні одиниці, в найпростіших реалізаціях є забороненим (оскільки вводить схему в режим генерації), в складніших реалізаціях RS-тригер переходить в третій стан $\boldsymbol{Q\bar{Q} = 0}$. Одночасне зняття двох «1» практично неможливе. При знятті однієї з «1» RS-тригер переходить в стан, що визначається другою «1». Таким чином RS-тригер має три стани, з яких два стійких (при знятті сигналів керування RS-тригер залишається у встановленому стані) і одне нестійке (при знятті сигналів керування RS-тригер не залишається у встановленому стані, а переходить в один з двох стійких станів).

RS-тригер використовується для створення сигналу з позитивним та негативним фронтами, окремо керованими за допомогою стробів, рознесених в часі. Також RS-тригери часто використовуються для запобігання так званого явища брязкоту контактів.

RS-тригери іноді називають RS-фіксаторами.

D-тригери

D-тригери також називають тригерами затримки (від англ. Delay).

D-тригер синхронний

D-тригер (D від англ. delay – затримка або від англ. data – дані) – запам'ятовує стан входу та видає його на вихід. D-тригери мають, як мінімум, два входи: інформаційний D і синхронізації C. Після приходу активного фронту імпульсу синхронізації на вхід C D-тригер відкривається. Збереження інформації в D-тригерах відбувається після спаду імпульсу синхронізації C. Оскільки інформація на виході залишається незмінною до приходу чергового імпульсу синхронізації, D-тригер називають також тригером із запам'ятовуванням інформації або тригером-засувкою. Міркуючи суто теоретично, парафазний (двофазний) D-тригер можна утворити з будь-яких RS- або JK-тригерів, якщо на їх входи одночасно подавати взаємно інверсні сигнали.

S	R	$Q(t)$	$\overline{Q}(t)$	$Q(t+1)$	$\overline{Q}(t+1)$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	не визначено	не визначено
1	1	1	0	не визначено	не визначено

Рисунок 1.16 – Таблиця істинності асинхронного RS-тригера

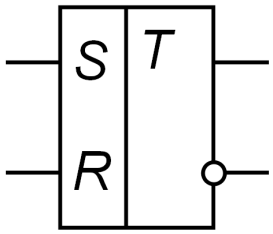


Рисунок 1.17 – Умовне графічне позначення асинхронного RS-тригера

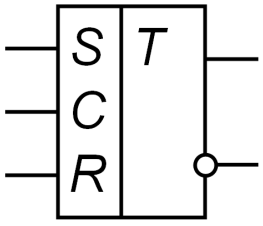


Рисунок 1.18 – Умовне графічне позначення синхронного RS-тригера

C	S	R	$Q(t)$	$Q(t+1)$
0	x	x	0	0
			1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	не визначено
1	1	1	1	не визначено

Рисунок 1.18 – Таблиця істинності синхронного RS-тригера

D-тригер переважно використовується для реалізації засувки. Наприклад, для зняття 32 біт інформації з паралельної шини, беруть 32 D-тригери і об’єднують їх входи синхронізації для керування записом інформації в засувку, а 32 входи D під’єднують до шини.

У одноступінчатих D-тригерах під час прозорості всі зміни інформації на вході D передаються на вихід Q. Там, де це небажано, потрібно застосовувати двоступеневі (двотактні, Master-Slave, MS) D-тригери.

D	Q(t)	Q(t+1)
0	0	0
0	1	0
1	0	1
1	1	1

Рисунок 1.19 – Таблиця істинності синхронного D-тригера із статичним входом синхронізації C.

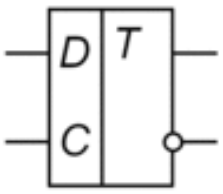


Рисунок 1.20 – Умовне графічне позначення синхронного D-тригера із статичним входом синхронізації C.

D-тригер двоступінчастий

У одноступінчастому тригері є одна щабель запам’ятовування інформації, а в двоступінчастому – два такі щаблі. Спочатку інформація

записується в першу сходинку, а потім переписується у другу та з’являється на виході. Двоступінчастий тригер позначають ТТ. Двоступінчатий D-тригер називають тригером з динамічним керуванням.

Т-тригери

Т-тригер (від англ. Toggle – перемикач) часто називають рахунковим тригером, оскільки він є найпростішим лічильником до 2.

Т-тригер асинхронний

Асинхронний Т-тригер не має входу дозволу рахунку – Т і переключається по кожному тактовому імпульсу на вході С.

Т-тригер синхронний

T	Q(t)	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

Рисунок 1.21 – Таблиця істинності синхронного Т-тригера з динамічним входом синхронізації С на схемах.

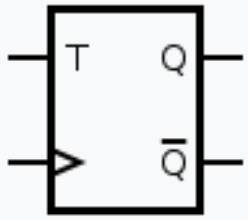


Рисунок 1.22 – Умовне графічне позначення синхронного Т-тригера з динамічним входом синхронізації С на схемах.

Синхронний **Т-тригер**, при одиниці на вході **Т**, по кожному такту на вході С змінює свій логічний стан на протилежний, і не змінює вихідний стан при нулі на вході **Т**. Т-тригер можна побудувати на JK-тригері, на двоступінчатому (Master-Slave, MS) D-тригері і на двох одноступінчатих D-тригерах та інверторі.

Як можна бачити в таблиці істинності JK-тригера, він переходить в інверсний стан щоразу при одночасній подачі на входи **J** і **K** логічної 1. Ця властивість дозволяє створити на базі JK-тригера Т-тригер, об’єднуючи входи **J** і **K**.

У двоступінчатому (Master-Slave, MS) D-тригері інверсний вихід Q з’єднується з входом D, а на вхід **C** подаються лічильні імпульси. Внаслідок цього тригер при кожному рахунковому імпульсі запам’ятовує значення Q, тобто буде перемикатися в протилежний стан.

Т-тригер часто застосовують для пониження частоти в 2 рази, при цьому на **T** вхід подають одиницю, а на **C**– сигнал з частотою, яка буде поділена на 2.

JK-тригер

J	K	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Рисунок 1.23 – Таблиця істинності JK-тригера з додатковими асинхронними інверсними входами **S** і **R**

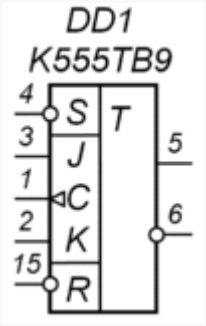


Рисунок 1.24 – Умовне графічне позначення JK-тригера з додатковими асинхронними інверсними входами **S** і **R**

JK-тригер працює так само як RS-тригер, з одним лише винятком: при подачі логічної одиниці на обидва входи J і K стан виходу тригера змінюється на протилежне. Вхід **J** (від англ. Jump – стрибок) аналогічний входу **S** у RS-тригера. Вхід **K** (від англ. Kill – вбити) аналогічний входу **R** у RS-тригера. При подачі одиниці на вхід **J** і нуля на вхід **K** вихідний стан тригера стає рівним логічній одиниці. А при подачі одиниці на вхід **K** і нуля на вхід **J** вихідний стан тригера стає рівним логічному нулю. JK-тригер на відмінну від RS-тригера не має заборонених станів на основних входах, проте це ніяк не допомагає при порушенні правил розробки логічних схем. На практиці застосовуються лише синхронні JK-тригери, тобто стани основних входів **J** і **K** враховуються лише в момент тактування, наприклад по позитивному фронту імпульсу на вході синхронізації.

На базі JK-тригера можна побудувати D-тригер або T-тригер. Як можна бачити з таблиці істинності JK-тригера, він переходить в інверсний стан щоразу

при одночасній подачі на входи **J** і **K** логічної 1. Ця властивість дозволяє створити на базі JK-тригера Т-тригер, об'єднавши входи **J** і **K** [2].

Алгоритм функціонування JK-тригера можна представити формулою:

$$Q(t+1) = \bar{Q}(t) \cdot J + Q(t) \cdot \bar{K}$$

1.3 Регістр

Регістр – послідовний або паралельний логічний пристрій, який виконує функцію прийому, запам'ятовування і передачі інформації.

Інформація в регістрі зберігається за видом числа (слова), зображеного комбінацією сигналів 0 і 1. Кожному розряду числа, що записаний в регістр, відповідає свій розряд, побудований, як правило, на базі тригерів RS-, D- або JK- типу.

На регістрах можна виконувати операції перетворення інформації з одного виду на інший, наприклад, послідовного коду на паралельний. Регістри можуть використовуватися для виконання деяких логічних операцій, наприклад, логічне порозрядне множення.

Класифікація регістрів

За способом запису і зчитування двійкової інформації.

Послідовні. В послідовних регістрах запис і зчитування інформації здійснюється послідовно за часом, тобто почергово. Вони мають послідовні виходи. Інформація записується шляхом послідовного зсуву числа синхроімпульсами. Тому регістри послідовного типу носять назву регістрів зсуву.

Паралельні. В паралельних регістрах, які мають паралельні входи та виходи, запис інформації виконуються одночасно в усіх розрядах за один такт керування. Такі регістри називають регістрами пам'яті.

Паралельно-послідовні. Паралельно-послідовні регістри мають або паралельний вхід та послідовний вихід, або послідовний вхід та паралельний вихід. В перших регістрах інформація записується одночасно по паралельних входах, а зчитується почергово, в других – записується почергово, а зчитується одночасно. Паралельно-послідовні регістри можуть бути як регістрами зсуву, так і регістрами пам'яті.

За способом приймання та передавання інформації.

Регістри типу SISO (англ. Serial In Serial Out) – з послідовним входом та послідовним виходом;

Регістри типу SIPO (англ. Serial In Parallel Out) – з послідовним входом та паралельним виходом;

Регістри типу PISO (англ. Parallel In Serial Out) – з паралельним входом та послідовним виходом;

Регістри типу PIPO (англ. Parallel In Parallel Out) – з паралельними входом та виходами.

Найбільш універсальними вважаються регістри, що мають у своєму складі одночасно послідовні і паралельні входи й виходи. Такі регістри називають регістрами з послідовно-паралельним прийманням інформації та послідовно-паралельним передаванням [3].

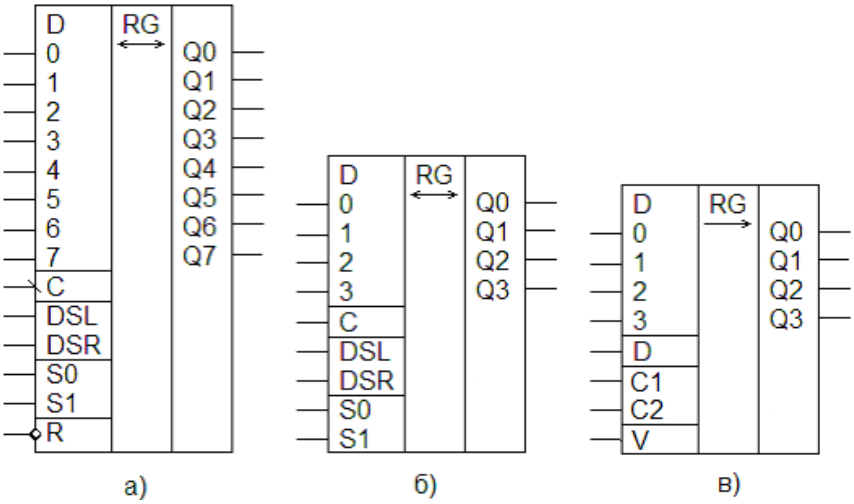


Рисунок 1.25 – Універсальні регістри зсуву: а – К155ІР13, б – К500ІР141, в – КМ155ІР1

1.4 Лічильник

Лічильник (counter) – пристрій для підрахунку кількості сигналів, що надходять на його вхід.

Двійкові лічильники реалізують лічбу вхідних імпульсів у двійковій системі числення.

Число розрядів n двійкового підсумовуючого лічильника для заданого модуля М знаходять із виразу $n = \log_2 M$. Значення поточного числа N+ вхідних імпульсів n-розрядного підсумовуючого лічильника при відліку з нульового початкового стану визначають за формулою:

$N+= Q_i=2_{n-1}\ Q_n+2_{n-2}\ Q_{n-1}+...+20\ Q_1$,

де 2_{i-1} – i-тий розряд; Q_iÎ{0,1} – логічне значення прямого виходу тригера i-го розряду. Розряди двійкового лічильника будуються на двоступеневих Т-тригерах або D-тригерах з динамічним керуванням по фронту синхросигналу (в лічильному режимі).

У двійковому підсумовуючому лічильнику перенесення P_i в сусідній старший розряд Q_{i+1} виникає в тому випадку, коли в момент надходження чергового лічильного імпульсу U+ всі молодші розряди знаходяться в одиничному стані, тобто $P_i=U+Q_iQ_{i-1}...Q_1=1$. Після вироблення перенесення старший розряд перемикається в стан «1», а всі молодші розряди – в стан «0».

Асинхронні підсумовуючі лічильники на двоступеневих Т-тригерах будуються так, щоб вхідні імпульси U+ надходили на лічильний вхід тільки першого (молодшого) розряду. Сигнали перенесення передаються асинхронно

(послідовно в часі) з прямих виходів молодших розрядів на Т-входи сусідніх старших.

Двійкові реверсивні лічильники. Двійкові реверсивні лічильники мають переходи у двох напрямках: в прямому (при лічбі підсумовуючих сигналів U+) і в зворотному (при переліку віднімальних сигналів U–). Поточне значення різниці підрахованих імпульсів визначається із співвідношення åU+ – åU– = N – N_n де N – значення коду на прямих виходах тригерів лічильника; N_n – попередньо записане в лічильник початкове число. Розрізняють одноканальні та двоканальні реверсивні лічильники. В одноканальних реверсивних лічильниках підсумовуючі U+ і віднімальні U– сигнали почергово надходять на спільний лічильний вхід, а напрямок лічби задається напрямком кіл міжрозрядних перенесень або позик. Для перемикання міжрозрядних зв'язків у одноканальному реверсивному лічильнику потрібні додаткові керуючі сигнали.

Двоканальні реверсивні лічильники мають два лічильних входи: один для підсумовуючих імпульсів U+, другий – для віднімальних U–. Перемикання ланцюгів міжрозрядних зв'язків здійснюється автоматично лічильними сигналами: для переносів – імпульсами U+, для позики – імпульсами U–. Для задання напрямку лічбі використовують додатковий RS-тригер: з його прямого виходу знімається сигнал керування додаванням УД (вмикає кола перенесення), а з інверсного виходу – сигнал керування відніманням УВ (вмикає кола позики).

Двійково-десяткові лічильники. Двійково-десяткові лічильники реалізують лічбу імпульсів у десятковій системі числення, причому кожна десяткова цифра від нуля до дев'яти кодується чотирирозрядним двійковим кодом (тетрадою). Ці лічильники часто називають десятковими або декадними, оскільки вони працюють з модулем лічби, кратним десяти (10, 100, 1000 і т.д.).

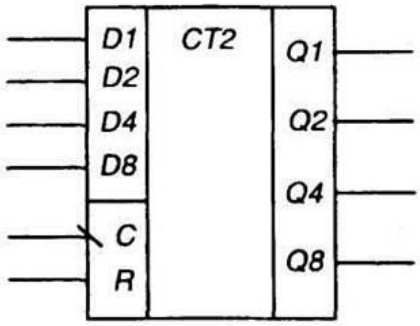


Рисунок 1.26 – Умовне позначення чотирирозрядного послідовного. двійкового лічильника

Декада будується на основі чотирирозрядного двійкового лічильника, в якому вилучається надлишкове число станів. Вилучення зайвих шести станів у декаді досягається багатьма способами: попереднім записуванням числа 6 (двійковий код 0110); після лічби дев'ятого імпульсу вихідний код дорівнює 1111 і десятковий сигнал повертає лічильник у початковий стан 0110, отже, тут результат лічби фіксується двійковим кодом з надлишком блокування переносів: лічба імпульсів до дев'яти здійснюється у двійковому коді, після чого вмикаються логічні зв'язки блокування перенесень; з надходженням

десятого імпульсу лічильник закінчує цикл роботи і повертається в початковий нульовий стан; введенням обернених зв'язків, які забезпечують лічбу в двійковому коді й примусовим перемиканням лічильника в нульовий початковий стан після надходження десятого імпульсу [4].

1.5 Шифратор

Шифратор (англ. Encoder) – логічний пристрій, що виконує логічну функцію перетворення *n*-розрядного коду в *k*-розрядний *m*-ковий (найчастіше двійковий) код.

Двійковий шифратор виконує логічну функцію перетворення *k*-того однозначного коду в двійковий.

Якщо кількість вхідних даних (входів) рівна кількості можливих комбінацій сигналів на виході, то такий шифратор називається повним, в іншому випадку – неповним.

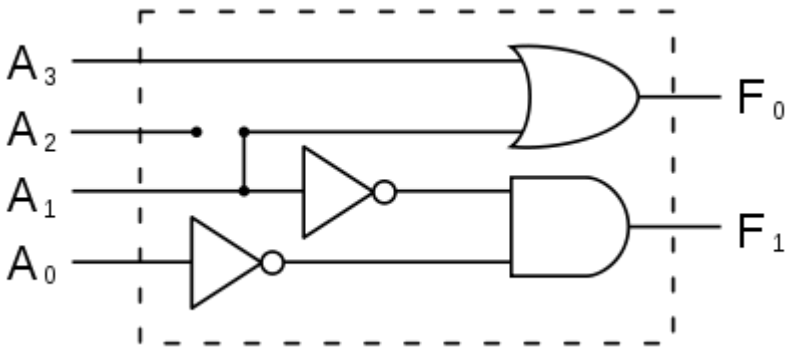


Рисунок 1.27 – Шифратор

Число входів і виходів в повному *k*-ковому шифраторі задається співвідношенням

$$n = 2^m,$$

де *n* – кількість входів,
m – кількість виходів [5].

1.6 Дешифратор

Дешифратор або декодер (англ. decoder) – логічний пристрій, який перетворює код числа, що поступило на вхід, в сигнал на одному з його виходів. Вихідними функціями дешифратора є різноманітні конституенти

одиниці: $\bar{Q}_0\bar{Q}_1 \dots Q_n, \bar{Q}_0\bar{Q}_1, \dots, \bar{Q}_{n-1}\bar{Q}_n, \dots, Q_0Q_1 \dots Q_n$. Якщо число представлено у вигляді n двійкових розрядів, то дешифратор повинен мати 2^n виходів. Дешифратор довільної складності може бути складено з трьох базових логічних елементів: кон'юнкції, диз'юнкції та заперечення.

Види дешифраторів

За принципом дії розрізняють такі види дешифраторів:

– послідовні;

– паралельні;

– паралельно-послідовні.

Розрізняють дешифратори першого та другого роду.

Дешифратори першого роду реалізують систему функцій, кожна з яких приймає одиничне значення при відповідному одиничному значенні вхідного слова.

Дешифратори другого роду реалізують систему функцій, кожна з яких приймає одиничне значення при визначених діапазонах вхідного слова.

За способом побудови розрізняють:

– лінійні дешифратори – n змінних, представляють сукупність не зв'язаних між собою 2^n систем збігу на n входів, кожна з яких реалізує відповідну конституенту одиниці;

– пірамідальні дешифратори – будуються за принципом послідовних каскадів: на першому каскаді реалізуються конституенти одиниці для двох змінних, на n–1 реалізуються конституенти одиниці для n змінних, при цьому, на вході отримується вихід з попереднього каскаду[6].

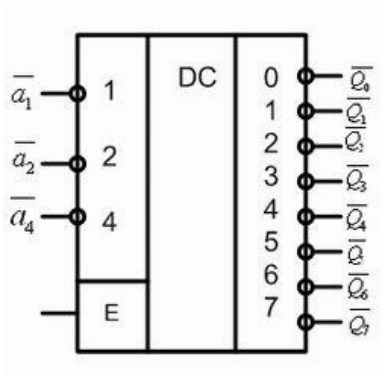


Рисунок 1.28 – Умовне позначення дешифратора 3 на 8 на схемах.

1.7 Мультиплексор

Мультиплексори відносяться до пристроїв комутування цифрової інформації. Вони здійснюють комутацію одного з декількох інформаційних входів X_i до одного виходу y. Мультиплексори мають декілька інформаційних входів, адресні входи, вхід дозволу мультиплексування (стробуючий вхід) та один вихід.

Кожному з інформаційних входів мультиплексора відповідає номер, який називається адресою, двійкове число якого подається до адресних входів.

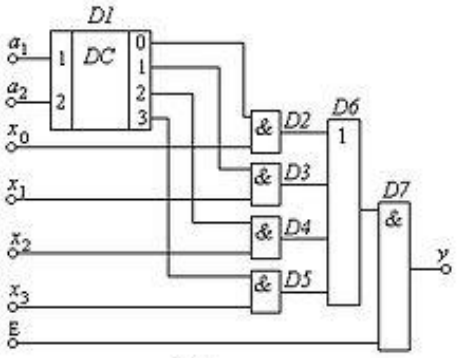
Число інформаційних входів $n_{\text{інф.}}$ і число адресних входів $n_{\text{адр.}}$ зв'язані співвідношенням: $n_{\text{інф}}=2^{n_{\text{адр}}}$.

Адресний дешифратор D1, перетворює двійковий код у десятковий для керування роботою мультиплексора. Залежно від комбінації стану адресних входів a_1 та a_2 на одному з чотирьох виходів дешифратора з'являється одиничний потенціал, який дає дозвіл на спрацьовування відповідної схеми І (D2…D5). Наприклад, при адресному числі 01, коли a_1= 1 та a_2= 0, на виході 1 дешифратора D1 установлюється рівень логічної одиниці, а на всіх останніх – нульовий. Тому логічний елемент D3 має дозвіл на спрацьовування.

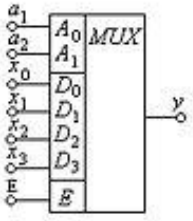
Якщо при цьому на інформаційному вході x_1 діє логічна одиниця, то на виході D3 установлюється 1, а при x_1=0 на виході логічного елемента D3 буде теж нульовий потенціал. При цьому, незалежно від стану інформаційних входів X0, X2, X3, на виході логічного елемента АБО D6 інформація повторює стан X1. Якщо активізований вхід дозволу E=1, то на виході мультиплексора *y* з'являється 1 або 0 залежно від значення X1.

Функціонування мультиплексора описується таблицею істинності [7]:

Адресні входи		Керуючий вхід E	Вхід *y*
a_1	a_2		
X	X	0	0
0	0	1	X_0
1	0	1	X_1
0	1	1	X_2
1	1	1	X_3



а)



б)

Рисунок 1.29 – Мультиплексор 4-1; а) – схема; б) – умовне позначення

1.8 Демультиплексор

Демультиплексор відноситься до пристроїв комутування цифрової інформації. Він здійснює комутацію одного інформаційного входу до одного з декількох виходів, адреса якого задана. Демультиплексор має один інформаційний вхід, декілька виходів та адресні входи.

Таким чином, на приймальному кінці мультиплексованої магістралі потрібно виконати зворотну операцію – демультиплексування. Демультиплексор можна реалізувати на дешифраторі з *n* входами, в якому вхід дозволу Е використовується як інформаційний. Якщо для побудови схеми демультиплексора використати дешифратор без входу дозволу Е, то необхідно мати *m* двовхідних логічних елементів 2І.

Входи дешифратора a_1, a_2 є адресними. Тому в залежності від адресного числа лише на одному з виходів дешифратора з'являється логічна одиниця, яка дає дозвіл до спрацювання лише одного з чотирьох кон'юкторів D2…D5. На другі входи кожного кон'юктора надходить шина сигналу *x*.

Вхідна інформація відтворюється на виході одного з чотирьох логічних елементів D2…D5, який одержав дозвіл по другому адресному входу.

Можна виконати синхронний демультиплексор, якщо використовувати тривходові логічні елементи 3І і на третій вхід подати синхросигнал або сигнал дозволу від зовнішнього джерела.

Функціонування демультиплексора 1-4 відображається таблицею істинності [8].

Адресні входи		Виходи			
a_1	a_2	y_0	y_1	y_2	y_3
0	0	*x*	0	0	0
1	0	0	*x*	0	0
0	1	0	0	*x*	0
1	1	0	0	0	*x*

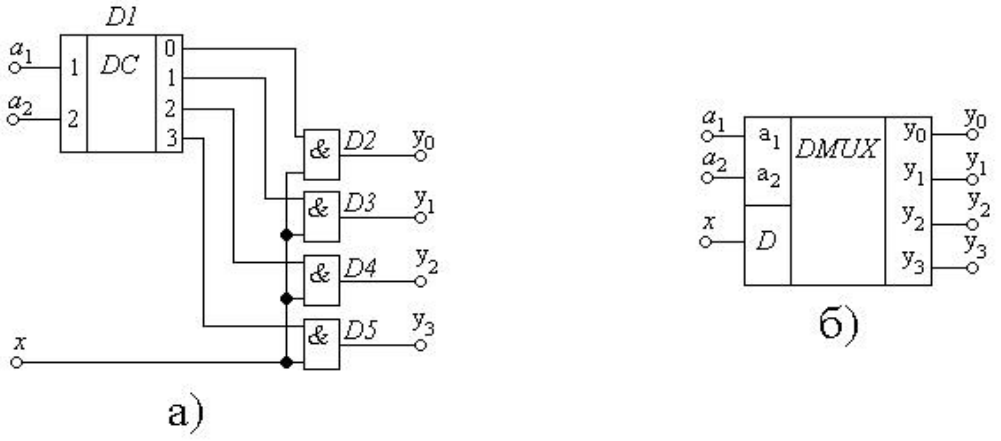


Рисунок 1.30 – Демультиплексор 1-4 на базі дешифратора D1 та логічних елементів 2І D2…D5 (без входу дозволу); а – схема; б – умовне позначення

1.9 Суматор

Суматор (англ. adder) – вузол ЕОМ, призначений для утворення суми двох операндів; цифрова схема, що виконує додавання чисел.

Суматором називається функціональний вузол комп'ютера, призначений для додавання двох *n*-розрядних слів (чисел). Операція віднімання замінюється додаванням слів в оберненому або доповнювальному кодах. Операції множення та ділення перетворюються на реалізації багаторазового додавання та зсуву. Тому суматор є важливою частиною арифметико-логічного пристрою. Функція суматора позначається літерами SM або Σ.

Суматор складається з окремих схем, які називаються однорозрядними суматорами; вони виконують усі дії з додавання значень однойменних розрядів двох чисел (операндів). Суматори класифікують за такими ознаками:

– способом додавання – паралельні, послідовні та паралельно-послідовні;

– кількістю вхідних клем – напівсуматори, однорозрядні або багаторозрядні суматори;

– організацією зберігання результату додавання – комбінаційні, накопичувальні, комбіновані;

– системою числення – позиційні (двійкові, двійково-десяткові, трійкові) та непозиційні, наприклад, у системі залишкових класів;

– розрядністю (довжиною) операндів – 8-, 16-, 32-, 64-розрядні;

– способом подання від'ємних чисел – в оберненому або доповнювальному кодах, а також їх модифікаціях;

– часом додавання – синхронні та асинхронні.

У паралельних *n*-розрядних суматорах значення всіх розрядів операндів надходять одночасно на відповідні входи однорозрядних підсумовуючих схем. У послідовних суматорах значення розрядів операндів та перенесення, які запам'ятовувалися в минулому такті, надходять послідовно в напрямку від молодших розрядів до старших на входи одного однорозрядного суматора. В паралельно-послідовних суматорах числа розбиваються на частини, наприклад, байти, розряди байтів надходять на входи восьмирозрядного суматора паралельно (одночасно), а самі байти – послідовно, в напрямку від молодших до старших байтів з врахуванням запам'ятованого перенесення.

У комбінаційних суматорах результат операції додавання запам'ятовується в регістр результату. В накопичувальних суматорах процес додавання поєднується зі зберіганням результату. Це пояснюється використанням Т-тригерів як однорозрядних схем додавання.

Організація перенесення практично визначає час виконання операції додавання. Послідовні перенесення схемно створюються просто, але є повільнодіючими. Паралельні перенесення схемно реалізуються значно складніше, але дають високу швидкодію.

Розрядність суматорів знаходиться в широкому діапазоні 4-16 – для мікро- та міні-комп'ютерів та 32-128 і більше – для універсальних машин.

Суматори з постійним інтервалом часу для додавання називаються синхронними. Суматори, в яких інтервал часу для додавання визначається моментом фактичного закінчення операції, називаються асинхронними. В асинхронних суматорах є спеціальні схеми, які визначають фактичний момент закінчення додавання і повідомляють про це в пристрій керування. На практиці переважно використовуються синхронні суматори.

Суматори характеризуються такими параметрами:

– швидкодією – часом виконання операції додавання tΣ, який відраховується від початку подачі операндів до одержання результату; нерідко швидкодія характеризується кількістю додавань в секунду FΣ = 1/ tΣ, тут розуміємо операції регістр-регістр (тобто числа зберігаються в регістрах АЛП);

– апаратурними затратами: вартість однорозрядної схеми додавання визначається загальною кількістю логічних входів використаних елементів;

– вартість багаторозрядного суматора визначається загальною кількістю використаних мікросхем;

– споживаною потужністю [9].

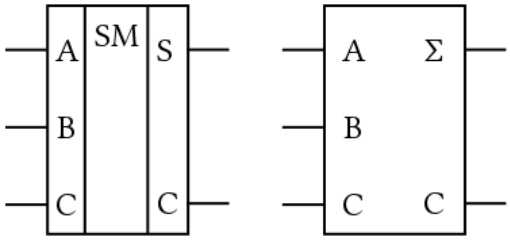


Рисунок 1.31 – Позначення суматора на електронних схемах

ЧАСТИНА ІІ ПРОГРАМУВАННЯ ЦІЛОЧИСЛОВОГО ПРОЦЕСОРА (CPU)

Мікропроцесор складається з трьох основних частин: пристрою обробки, пристрою управління пам'яттю, інтерфейсного блоку.

Пристрій обробки складається з виконавчого пристрою (операційної частини) і блоку команд (керуючої частини). Містить 8 32-х розрядних регістрів загального призначення, 64-х бітовий циклічний зсувач. Множення і ділення здійснюється на 1 біт за цикл. Алгоритм множення такий, що процес припиняється, коли найбільш значущий біт, множиться на всі нулі. Типовий час множення 32-х розрядних чисел близько 1 мкс (для процесора і 80386).

Пристрій управління пам'яттю складається з сегментного і сторінкового блоків. Сегментний блок дозволяє працювати з логічними адресами. Сторінкова організація використовується всередині сегмента і керує фізичними адресами. Кожна задача може мати до 16381 (2^{14}) сегменти до 4 Гбайт кожен (2^{32}), тобто віртуальна пам'ять може бути розміром 64 Тбайт (2^{46}).

Інтерфейсний блок забезпечує взаємодію з зовнішніми пристроями, включаючи автоматичне керування розрядністю шини, і формування сигналів активності байтів.

Мікропроцесори 386+ можуть функціонувати в трьох режимах:

– REAL ADDRESS MODE – режим реальної адресації (РРА) – характеризується тим, що мікропроцесор працює як дуже швидкий 8086 з 32-бітовим розширенням; у цьому режимі можлива адресація 1 Мбайт фізичної пам'яті (насправді, як у і 80286, – майже на 64 Кбайта більше);

– PROTECTED ADDRESS MODE – режим захищеної віртуальної адресації (РВА) – реалізує всі переваги мікропроцесора (режим паралельного виконання кількох задач кількома 8086 – по одному на завдання). На одному процесорі в такому режимі можуть одночасно виконуватися кілька завдань з ізольованими один від одного реальними ресурсами. При цьому використання фізичного адресного простору пам'яті управляється механізмами сегментації і трансляції сторінок. Спроби виконання неприпустимих команд, виходу за рамки відведеного простору пам'яті і дозволеної області вводу-виводу контролюються системою захисту.

– VIRTUAL 8086 MODE – режим віртуального процесора 8086 (скорочено – V86). Прикладна програма, яка виконується в цьому режимі, вважає, що вона працює на процесорі 8086. Однак, деякі команди, переважно пов'язані з управлінням введенням-виведенням, програмі виконувати забороняється, тому при порушенні захисту генерується переривання і управління передається операційній системі.

2.1 Програмна модель 32-х розрядних процесорів

Мікропроцесор 386+ має 31 регістр (у PENTIUM+ – 32 регістра), розбиті на наступні групи:

– регістри загального призначення;
– сегментні регістри;
– вказівник команд і регістр прапорів (ознак);
– керуючі регістри;
– регістри системних адрес;
– налагодження регістри;
– тестові регістри.

Набір ***регістрів загального призначення*** (рис. 2.1) включає відповідні регістри процесорів і 8086 та і 80286. Усі ці регістри, крім сегментних, мають розрядність 32 біти і до попереднього позначення їх імен додалася попереду літера «E» (Extended – розширений). Відсутність літери «E» у назві означає посилання на молодші 16 біт розширених регістрів. Звернутися до старших 16-ти біт розширених регістрів жодна команда не може. Як і в і 8086, можливе незалежне звернення до молодшого і старшого байтів регістрів AX, BX, CX, DX.

Рисунок 2.1 – Регістри 32-х розрядних мікропроцесорів (386+)

Архітектура мікропроцесора 386+ дозволяє безпосередньо звертатися до 6 сегментів (розміром до 4 Гб кожен) за допомогою спеціальних селекторів, що завантажуються в сегментні регістри програмно. Вміст регістрів загального призначення, селекторів, покажчика команд і регістра прапорів (ознак) залежить від задачі, що виконується і автоматично перевантажується при перемиканні задач.

Інші регістри мікропроцесора використовуються, переважно, для спрощення проектування та налагодження операційної системи.

Регістри загального призначення використовуються для зберігання операндів і адрес. Можуть працювати з операндами, що мають довжину 1, 8, 16, 32 і 64 біти або з бітовими полями довжиною від 1 до 32 біт.

Вказівник команд **EIP** зберігає зміщення, яке завжди складається зі значенням кодового сегментного регістра (CS) і визначає адресу наступної команди. При 16-ти бітовій адресації використовуються тільки молодші 16 біт (IP).

Регістр прапорів (ознак) EFLAGS (рис. 2.2) відображає стан мікропроцесора. При використанні тільки 16-ти молодших розрядів регістр прапорів сумісний з попередніми моделями мікропроцесорів.

31																16	15															0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF
										x	x	x	x	x	x		x	x	x	x	c	x	x	s	s		s		s		s

Рисунок 2.2 – Регістр прапорів EFLAGS (ознак)

На рис. 2.2 позначені символами: x – системний прапор; s – прапор стану; c – керуючий прапор.

Призначення біт регістра прапорів (ознак)

CF (Carry Flag) – прапор переносу, що показує перенесення (позику) зі старшого біта при арифметичних операціях, а також значення висунутого біта при зсуві операнда;

AF (Auxiliary Flag) – прапор допоміжного переносу (позики) в молодшій тетраді для десяткової арифметики;

OF (Overflow Flag) – прапор арифметичного переповнення, визначає (при OF=1) вихід знакового результату за межі діапазону;

ZF (Zero Flag) – прапор нуля, показує (при ZF=1) нульовий результат команди;

SF (Sign Flag) – прапор знаку, дублює значення старшого біта результату, який при використанні додаткового коду відповідає знаку числа;

PF (Parity Flag) – прапор паритету (парності), що фіксує (при PF=1) наявність в молодшому байті результату парного числа одиничних бітів.

IOPL (Input/Output Privilege Level) – використовується тільки в режимі віртуальної адресації. IOPL вказує максимальну величину поточного пріоритету, що забезпечує виконання команд вводу-виводу без реакції на 13

помилку. Ця ознака також забезпечує вибір IF, коли нове значення виштовхується з стека в регістр ознак. POPF і IRET можуть змінювати поле IOPL, коли IOPL=0 (CPL=0). При перемиканні задач IOPL може змінюватися завжди при перепису TSS (286+).

NT (Nested Task Flag) – прапор вкладеної задачі (286+);

ID (Id Flag) – прапор доступності команди ідентифікації CPUID (PENTIUM+ і деякі 486+);

VIP (Virtual Intterruptt Pending) – віртуальний запит переривання (PENTIUM+);

VIF (Virtual Interrupt Flag) – віртуальна версія прапора IF (дозвіл переривання) для багатозадачних систем (PENTIUM+).

AC (Alignment Check) – прапор контролю вирівнювання. При виконанні програм на рівні привілеїв 3 у разі звертання до операнду, не вирівняному на відповідній межі (2, 4, 8 байт), і при встановленому прапорі AC відбудеться виключення-відмова 17 з нульовим кодом помилки. На рівнях привілеїв 0, 1, 2 контроль вирівнювання не проводиться (486+).

VM (Virtual 8086 Mode) – забезпечує режим віртуального 8086 всередині режиму віртуальної адресації. При VM = 1 мікропроцесор буде переключено в режим віртуального і 8086, при цьому управління перезавантаженням сегментів буде здійснюватися подібне до і 8086, але з виключенням 13 недійсних привілейованих команд. VM може бути встановлений в режимі віртуальної адресації командою IRET (якщо рівень пріоритету рівний 0) і завдання переключається на нижчий рівень. Команда POPF не впливає на VM. Команда PUSHF завжди скидає VM в 0, якщо вона виконується в режимі віртуального 8086. Вміст регістра ознак буде копіюватися при перериваннях або зберігатися при перемиканні завдання, якщо переривання буде виконуватися в режимі віртуального 8086 (386+).

RF (Resume Flag) – прапор поновлення, використовується спільно з налагоджувальними регістрами контрольних точок (переривань) або покрокового режиму. З його допомогою перевіряється хід виконання команд в налагоджувальному режимі (процес налагодження). Якщо встановлений RF=1, то це дозволяє ігнорувати помилки, що виникають при налагодженні до наступної команди. RF автоматично скидається в 0 при успішному виконанні команди (помилки не виявлені), за винятком команд IRET і POPF, а також JMP, CALL і INT при перемиканні задач. Ці команди встановлюють RF у стан, обумовлений станом пам’яті. Наприклад, наприкінці виконання підпрограми обслуговування контрольної точки команда IRET може встановити RF в стан, що відповідає значенням регістра ознак, що зберігається в стеку без повторної установки RF в 1 (386+).

NT (Nested Task Flag) – прапор вкладеної задачі (гніздування) використовується тільки в режимі віртуальної адреси. NT=1 вказує, що поточна задача є вкладеною відносно іншої задачі. Цей біт встановлюється і скидається при виклику інших задач. NT перевіряється командою IRET для визначення всередині заданого або зовнішнього відносно до даної задачі повернення. Команди POPF і IRET будуть встановлювати NT відповідно до того, що зберігається в стеку для будь-якого рівня привілейованості (286+).

Мікропроцесори 386+ містять 6 16-ти бітових ***сегментних регістрів*** (у попередніх поколінь – тільки 4 сегментних регістра, що зберігають значення селектора і визначають значення початкових (базових) адрес сегментів. У режимі віртуальної адресації кожен сегмент може змінюватися в діапазоні від одного байта до максимального значення фізичного адресного простору 4 Гбайти. В режимі реальної адресації розміри сегмента обмежені розміром 64 Кбайт.

Дескрипторні регістри сегментів програмно не видимі, але вони нерозривно пов'язані з відповідними сегментними регістрами (рис. 2.3). Кожен дескрипторний регістр зберігає 32-х бітову базову адресу сегмента, 20-ти бітовий розмір сегмента та інші необхідні його атрибути.

Сегментні регістри		Дескрипторні регістри – програмно недоступні (завантажуються автоматично)		
15 0		Базові адреси сегментів	Розміри сегментів	Атрибути сегментів
Селектор	CS			
Селектор	SS			
Селектор	DS			
Селектор	ES			
Селектор	FS			
Селектор	GS			

Рисунок 2.3 – Сегментні регістри і відповідні дескрипторні регістри мікропроцесора 386+

Коли значення селектора завантажується в сегментний регістр, у ***режимі віртуальної адресації*** відповідний дескрипторний регістр автоматично завантажується інформацією з дескрипторної таблиці.

В режимі віртуальної адресації базову адресу, розмір і атрибути сегментного дескриптора визначаються селектором. 32-х бітова ***базова адреса*** сегмента стає компонентом формування виконавчої адреси, 20-ти бітовий ***розмір сегменту*** використовується для перевірки меж робочої області, а ***атрибути*** перевіряються на відповідність типу запитуваної пам'яті (типу звертання).

В режимі реальної адресації безпосередньо використовується тільки базова адреса (зі зміщенням на 4 розряди ліворуч), а розміри сегмента та атрибути постійні (фіксовані для режиму реальної адресації).

2.2 Типи даних 32-х бітових процесорів

32-х розрядні процесори фірми INTEL (386+) працюють з цілими двійковими числами довжиною 8, 16 або 32 біта і двійково-кодованими

десятковими числами (BCD-числами) довжиною 8 біт. Двійкові числа допускають інтерпретацію як цілих без знака і цілих зі знаком чисел, а десяткові (BCD) – знака не мають.

У двійкових цілих числах без знака всі розряди вважаються значущими (див. рис. 4). ***Двійкові цілі числа зі знаком*** подаються у додатковому коді. Старший біт є знаковим (рис. 4): S = 0 – число додатне, S = 1 – число від'ємне.

Десяткові числа подаються в упакованому і неупакованому форматах. Упакований формат передбачає, що байт містить дві десяткові цифри в коді з вагами 8421, що займають молодшу і старшу тетради. Діапазон BCD-чисел, що подаються – 0…99 (рис. 4). У неупакованому форматі байт містить одну десяткову цифру, яка зазвичай зображується в символьному коді ASCII.

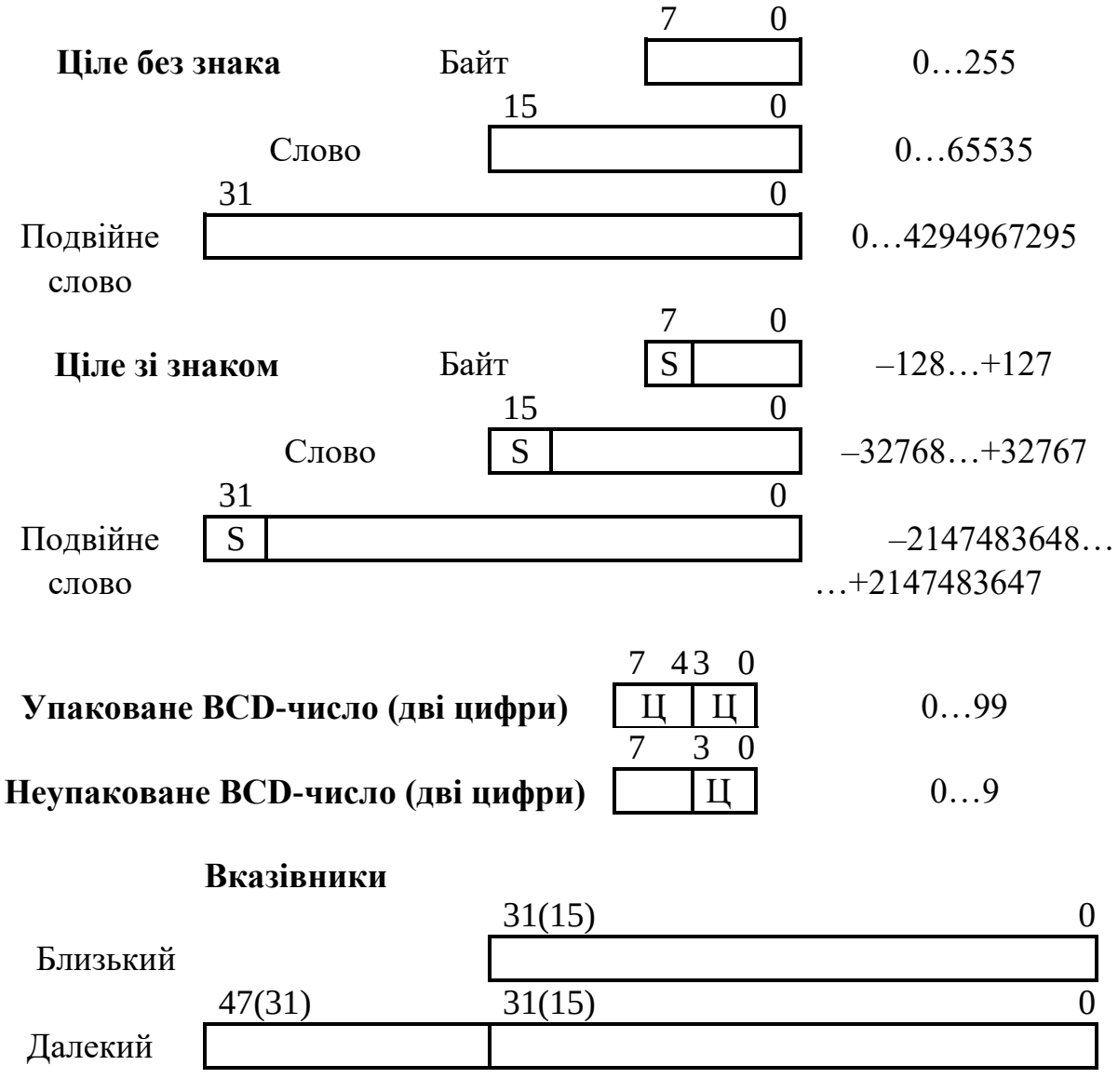


Рисунок 2.4 – Типи даних 32-х розрядних процесорів

Нові команди процесорів 386+ підтримують ***бітові дані***:
– ***біт*** – одиночний двійковий розряд;
– ***бітове поле*** – група до 32-х бітів;

– ***ланцюжок бітів (рядок)*** – набір послідовних бітів, довжиною до 4 Гбіт.

Процесор може легко оперувати з ланцюжками бітів, байтів, слів і подвійних слів. Під ***ланцюжком*** (string) слід розуміти послідовність практично будь-якої довжини окремих, але взаємопов'язаних елементів даних, що ***зберігаються за сусідніми адресами***.

Вказівники застосовуються для звернення до деяких об'єктів у пам'яті, наприклад, до адрес підпрограм. Близькі (NEAR) або внутрішньосегментний вказівник (див. рис. 2.4) – це 16-ти бітове або 32-х бітове зміщення усередині поточного сегмента. Далекий (FAR) або міжсегментний вказівник застосовується в тих випадках, коли програма здійснює передачу управління в інший сегмент. Такий вказівник визначає новий сегмент (за допомогою селектора) і 16-ти або 32-х бітове зміщення всередині цього сегменту.

При розміщенні операндів у пам'яті необхідно враховувати, що процесори 386+ не накладають обмеження на розміщення даних. Однак продуктивність процесора підвищується, якщо слова розміщені за парними адресами, а подвійні слова – за адресами, кратними чотирьом. Такий принцип називається ***вирівнювання адрес*** за межі слів і подвійних слів. Вирівнювання особливо важливо для стека, який працює тільки зі словами або подвійними словами.

2.3 Система команд процесорів 386+

Система включає 9 груп команд:

1. передачі даних;
2. арифметичні та логічні;
3. зсуву (зміщення);
4. обробки рядків;
5. маніпуляції бітами;
6. передачі управління;
7. підтримки мов високого рівня;
8. підтримки операційної системи;
9. керування процесором.

Команди можуть містити від 0 до 3 операндів, розміщених в регістрах, пам'яті або безпосередньо в команді. Більшість безоперандних команд – однобайтові. Одноопернадні команди зазвичай – двобайтові. Середня довжина команди – 3,2 байти. Це дозволяє зберігати в середньому 5 команд у 16-ти байтовій ***черзі команд блоку випереджальної вибірки***.

При використанні двох операндів можливі наступні типи взаємодії:

– регістр – регістр;
– пам'ять – регістр;
– регістр – пам'ять;
– безпосередній операнд – регістр;

– безпосередній операнд – пам’ять;

– пам’ять – пам’ять.

Операнди можуть бути 8, 16 або 32-х розрядними. Коли виконуються команди, написані для 386+, операнди мають довжину 8 або 32 біти, коли – для 80286 і 8086 – операнди 8 або 16 біт. До всіх інструкцій можуть додаватися префікси, які змінюють довжину операндів (тобто дозволяють використовувати 32-х бітові операнди в 16-ти бітових командах або 16-бітові операнди в 32-х бітових командах).

2.3.1 Режими (методи) адресації. Процесори 386+ забезпечують 13 режимів адресації, що розраховані на ефективне виконання програм, написаних на мовах високого рівня типу: C++, Фортран та ін..

Неявна адресація. Операнд адресується неявно, якщо в команді немає спеціальних полів для його визначення, тобто операнд задається полем команди. В асемблерних кодах з неявною адресацією поле операнда порожнє. Приклади команд з неявною адресацією:

```
AAA     ; Корекція регістра AL після додавання
CMC     ; Інверсія прапора перенесення
STD     ; Встановити в 1 прапор напрямку.
```

Режим регістрової адресації і ***режим безпосередньої адресації*** призначені, відповідно, для адресації одного з регістрів регістрового блоку або безпосереднього операнда в команді з розрядністю 8, 16 або 32 біти:

```
INC     esi     ; Інкремент регістра ESI
SUB     ECX, ECX        ; Скинути регістр ECX
MOV     EAX, CR0        ; Передати в EAX зміст CR0.
MOV     EAX, 0F0F0F0F0h     ; Завантажити константу в EAX
AND     AL, 0FH     ; Виділити молодшу тетраду регістра AL
BT      EDI, 3      ; Передати в прапор CF третій біт регістра EDI
```

Є 10 режимів ***адресації пам’яті***. Виконавча адреса включає в себе два компоненти адреси комірки пам’яті – сегмент і ефективна адреса (внутрішньосегментне зміщення). ***Ефективна адреса*** (EA) обчислюється додаванням наступних елементів:

– ***зміщення (відхилення)*** – ціла 8-ми або 32-х бітова величина зі знаком, безпосередньо задається в команді (16-бітові відхилення можуть використовуватися за допомогою префікса);

– ***база*** – вміст будь-яких регістрів загального призначення. Базові регістри зазвичай використовуються компіляторами як точки відліку локальної області пам’яті;

– ***індекс*** – вміст будь-яких регістрів загального призначення, крім ESP. Індексні регістри використовуються для доступу до елементів рядків або масивів.

– ***множник*** *f* вказує крок (1, 2, 4 або 8) для індексного регістра. Крок індексації дозволяє успішно адресувати масиви або структури, що містять багатобайтні операнди.

EA = БАЗА + ІНДЕКС * (КРОК ІНДЕКСАЦІЇ) + ВІДХИЛЕННЯ

Обчислення ефективної адреси практично не погіршує продуктивності процесора із-за використання конвеєрного режиму.

2.3.2 Режими адресації пам'яті. ***Пряма адресація*** – зсув (відхилення) адреси операнду міститься в 8, 16 або 32 розрядах команди:

```
MOV     AL, [2000h]          ; Передати байт у регістр AL
INC     dwordprt [123456h] ; Інкремент подвійного слова в пам'яті.
```

Регістровий непрямий метод адресації – базовий та індексний регістр містить адресу операнду:

```
MOV     AL, [ECX]          ; Передати в AL байт за адресою з ECX.
DEC     wordptr [ESI]      ; Декремент слова за адресою ESI.
```

Базова адресація – базовий регістр підсумовується з відхиленням:

```
MOV     EAX, [EBX+4]    ; Передати подвійне слово з пам'яті.
ADD     [ECX+10h], DX   ; Додати до слова у пам'яті.
```

Індексна адресація – індексний регістр (будь-який регістр загального призначення крім ESP) підсумовується з відхиленням:

```
SUB     array [ESI], 2      ; Відняти 2 з елемента масиву
IMUL    vector [ECX]        ; Помножити EAX на елемент масиву.
```

Індексна адресація з кроком – вміст індексного регістра множиться на крок «f» і підсумовується з відхиленням:

```
MOV     EAX, vec [ECX*4]        ; Переслати в EAX подвійне слово з масиву.
```

Базово-індексна адресація.

```
EA = БАЗА + ІНДЕКС:
ADD     EAX, [EBX][ESI]         ; Додати до EAX подвійне слово з пам'яті.
```

Базово-індексна адресація з кроком.

```
EA = БАЗА + ІНДЕКС * КРОК:
INC     wordptr [EDX][EDI*4]          ; Інкремент комірки пам'яті.
```

Базово-індексна адресація з відхиленням.

```
EA = БАЗА + ІНДЕКС + ВІДХИЛЕННЯ:
MOV     AX, [ECX][ESI+20h]    ; Переслати слово з пам'яті.
```

Базово-індексна адресація з відхиленням і з кроком.

```
EA = БАЗА + ІНДЕКС * КРОК + ВІДХИЛЕННЯ:
ADD     AX, [EDX][EDI*4+10h]        ; Скласти AX з коміркою пам'яті.
```

Стекова адресація (можна розглядати як варіант регістрової непрямої адресації) – у вказівнику стека ESP (SP) формується 32-х бітове (16-ти бітове) внутрішньосегментне зміщення для операнда в стековому сегменті:

```
PUSH    ECX         ; Включити в стек вміст регістра
PUSHFD              ; Включити в стек вміст EFLAGS
PUSH    4000h       ; Включити в стек константу
POP     EDX         ; Витягти з стека в регістр
POPFD               ; Витягти з стека до регістру EFLAGS
POP     [ESI]       ; Витягти з стека в комірку пам'яті.
```

У таблиці 1 показана різниця у використанні базових і індексних регістрів для 16-ти і 32-х бітових адрес.

Для забезпечення сумісності програмного забезпечення процесорів необхідно програми (з 16-бітовими командами мікропроцесорів 86 і 286) виконувати на мікропроцесорах 386+ в реальному або захищеному режимах. Процесор визначає розмірність адреси, аналізуючи біт **D** (Default) в дескрипторі сегмента. Якщо D=0, то всі довжини операнда і ефективних адрес становлять 16 біт. Якщо D=1, – 32 біта. В реальному режимі – 16 біт.

Зміна розмірності адреси і даних, що задаються бітом **D**, забезпечують два префікса, що обираються перед командами:

– ***префікс розмірності операндів*** (OperandSize),

– ***префікс довжини адреси*** (AddressSize).

Наявність префікса комутує (перемикає) розмір операнда або розмір ефективної адреси на значення, протилежне до того, що приймається за замовчуванням (за бітом **D**).

Префікси можуть використовуватися разом з будь-якою інструкцією і в будь-якому режимі – реальному, віртуальному та V86. Префікс довжини адреси не забезпечує розмірність адреси понад 64 Кбайти в режимі реальної адресації. Адреса понад 0FFFFh буде розглядатися як помилка.

Таблиця 2.1 – Базові та індексні регістри для 16-ти і 32-х бітових адрес

	16-ти бітова адреса	32-х бітова адреса
Базовий регістр	BX, BP	Будь-який 32-х бітовий режим загального призначення
Індексний регістр	SI, DI	Будь-який 32-х бітовий режим загального призначення, крім ESP
Крок індексації «f»	немає	1, 2, 4, 8
Зміщення	0, 8, 16 біт	0, 8, 32 біт

2.3.3 Використання сегментних регістрів. Основна структура в організації пам’яті – ***сегмент***.

Сегменти – блоки пам’яті змінної довжини (від 1 байта до 4 Гбайт), що мають певні атрибути. Три основних типи сегментів – ***стек, команди, дані***.

Для компактного кодування команд і підвищення продуктивності мікропроцесора команди не містять явної вказівки на сегментний регістр, що використовується. Визначення сегментного регістра (за замовчуванням) проводиться автоматично у відповідності з табл. 2.1. Сегментні регістри **FS *і* GS – *не вибираються за замовчуванням в жодній команді*** і можуть бути обрані тільки префіксом заміни сегмента.

Зазвичай назва сегментного регістра вказує на тип інформації, для адресації якої він використовується. Застосування префікса переадресації

дозволяє явно визначати сегментний регістр, що використовується (див. назву регістрів в дужках у другій колонці табл. 2.2), зокрема **FS** і **GS**.

Таблиця 2.2 – Вибір сегментних регістрів і внутрішньосегментного зміщення

Тип звертання до пам'яті	Сегментний регістр	Зміщення
Вибірка команди	CS	EIP (IP)
Звернення до стеку	SS	ESP (SP)
Адресація операнда	DS (CS, SS, ES, FS, GS)	EA
Елемент ланцюжка-джерела	DS (CS, SS, ES, FS, GS)	ESI (SI)
Елемент ланцюжка-приймача	ES	EDI (DI)
Операнд з використанням в якості базового регістра EBP (BP) або ESP (SP)	DS (CS, SS, ES, FS, GS)	EA

2.4 Команди передачі даних процесорів 386+

Команди цієї групи призначені для пересилок байтів (позначається B), слів (W) або подвійних слів (D) з пам'яті в регістр, з регістра в пам'ять і з регістру в регістр. В одній команді неможливо використання двох операндів, розташованих у пам'яті (за винятком ланцюгових команд і операцій зі стеком).

Команда **MOV** передає байт, слово або подвійне слово з джерела у приймач. В полі операндів приймач знаходиться на першому місці, джерело – на другому.

Команда **XCHG** здійснює обмін байтів, слів і подвійних слів. Відмінностей між приймачем і джерелом немає.

Команда **XLAT** замінює значення в регістрі AL на байт з таблиці, що адресується регістром (E)BX, причому індексом таблиці служить вміст регістра AL. Ця команда зручна для перетворення з одного коду в інший.

Команда **LEA** забезпечує обчислення ефективної адреси EA комірки пам'яті у відповідності із зазначеним способом адресації і завантаження EA (а не вмісту адресної комірки пам'яті!) у зазначений загальний регістр.

Команди **LDS, LES...** завантажують чотири (або шість) суміжних байта з пам'яті в адресований регістр (16 чи 32 біта) і у відповідний сегментний регістр (16 біт). Слово (подвійне слово) операнда джерела з комірки пам'яті, адресованої згідно із зазначеним методом адресації, передається в обраний регістр, а наступне слово – в регістр DS (команда LDS), в регістр ES (команда LES) і т. д.

У таблицях використовуються наступні позначення:

– src – операнд-джерело;

– dest – операнд-призначення (операнд-приймач);

– reg – 8/16/32-х бітовий регістр;

– reg16/32 – 16/32-х бітовий регістр;

– reg16 – тільки 16-ти бітовий регістр;

– reg32 – тільки 32-х бітовий регістр;

– mem – 8/16/32-х бітова комірка пам’яті, що адресується регістрами процесора;

Таблиця 2.3 – Команди пересилання даних

MOV dest, src	Пересилання (копіювання) даних з регістра, пам’яті або безпосереднього операнда в регістр або пам’ять
XCHG r/m, reg	Обмін даними (взаємний) між регістрами або регістром і пам’яттю
BSWAP reg32	Перестановка байтів в регістрі з порядку молодший-старший в порядок старший-молодший (486+)
MOVSXB reg, r/m	Копіювання байта з розширенням до слова або подвійного слова, заповнюючи старші біти знаком (386+)
MOVSXW reg, r/m	Копіювання слова з розширенням до подвійного слова, заповнюючи старші біти знаком (386+)
MOVZXB reg, r/m	Копіювання байта з розширенням до слова або подвійного слова, заповнюючи старші біти нулем (386+)
MOVZXB reg, r/m	Копіювання слова з розширенням до подвійного слова, заповнюючи старші біти нулем (386+)
XLAT	Трансляція (перекодування) змісту AL в значення з таблиці трансляції, що адресується в (E)BX: AL ← [(E)BX+AL]
LEA reg16/32, mem	Завантаження ефективної адреси в регістр
LDS reg16/32, mem	Завантаження в регістр (подвійного) слова з пам’яті, а в DS – наступного 16-бітового слова
LES reg16/32, mem	Завантаження в регістр (подвійного) слова з пам’яті, а в ES – наступного 16-бітового слова
LFS reg16/32, mem	Завантаження в регістр (подвійного) слова з пам’яті, а в FS – наступного 16-бітового слова
LGS reg16/32, mem	Завантаження в регістр (подвійного) слова з пам’яті, а в GS – наступного 16-бітового слова
LSS reg16/32, mem	Завантаження в регістр (подвійного) слова з пам’яті, а в SS – наступного 16-бітового слова
IN AL(AX), port8	Введення в AL (або AX, EAX) з порту з адресою port8
IN AL(AX), DX	Введення в AL (або AX, EAX) з порту з адресою, що зберігається в DX
OUT port8, AL(AX)	Виведення з AL (або AX, EAX) в порт з адресою port8
OUT DX, AL(AX)	Виведення з AL (або AX, EAX) в порт з адресою, що зберігається в DX

– r/m – 8/16/32-х бітовий регістр або комірка пам’яті, що адресується регістрами процесора;

– r/m/i – 8/16/32-х бітовий регістр, комірка пам’яті, що адресується регістрами процесора або безпосередній операнд;

– addr – 16/32-х бітова адреса;

– immed – безпосередній операнд.

Таблиця 2.4 – Команди роботи зі стеком

PUSH r/m	Поміщення (подвійного) слова з регістра або пам’яті в стек
PUSHimmed	Поміщення безпосереднього операнда в стек (286+)
PUSHA (D)	Поміщення в стек регістрів AX, CX, DX, BX, SP, BP, SI, DI (286+) або їх 32-х бітових розширень (386+)
POP r/m	Добування (подвійного) слова даних з стека в регістр або пам’ять
POPA (D)	Добування даних зі стеку в регістри DI, SI, BP, SP, BX, DX, CX, AX (286+) або їх 32-х бітових розширень (386+)
PUSHF (D)	Поміщення в стек регістра прапорів FLAGS (EFLAGS)
POPF (D)	Добування даних зі стеку в регістр прапорів FLAGS (EFLAGS)

Команда **PUSH** передає слово (або подвійне слово) з джерела в стек, а команда **POP** здійснює протилежну дію – передає (подвійне) слово з стека в приймач. ***Стек*** – це область пам’яті, в якій розміщується поточний сегмент стека. Регістр (E)SP містить зміщення останнього включеного в стек слова; воно (зміщення) називається ***вершиною стека***. У процесі включення в стек нових слів вони розташовуються за меншими адресами пам’яті; кажуть, що стек росте у напрямі зменшення адрес.

Команда **PUSH** починається зі зменшення (декремента) вмісту регістра (E)SP на 2 (або 4), тобто адресує наступне вільне слово (або подвійне слово) в стеку; після чого передається (подвійне) слово з джерела.

Команда **POP** передає слово (або подвійне слово) з стека в приймач і завершується збільшенням (інкрементом) вмісту (E)SP на 2 (або на 4).

Команда **PUSHA (D)** включає в стек регістри в такому порядку: (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, (E)DI. Включається те значенням регістра (E)SP, яке було в ньому до виконання команди PUSHA (D). При виконанні команди PUSHA (D) відбувається декремент вмісту регістра (E)SP на 2 (або на 4) при включенні в стек кожного регістру.

Добування з стека, що реалізовується командою **POPA (D)**, викличе інкремент вмісту регістра (E)SP на ту ж величину, тому команді POPA (D) не потрібен запам’ятований в стеку вміст регістра (E)SP.

Відмінність між знаковими і беззнаковими числами при виконанні арифметичних операцій полягає в інтерпретації двійкових наборів. Беззнакові числа – це звичайні двійкові числа (всі біти значущі), а знакові числа подані в додатковому коді.

Таблиця 2.5 – Команди цілочисельної арифметики

ADD r/m, r/m/i	Додавання двох операндів: r/m ← (r/m + r/m/i)
XADD r/m, reg	Обмін і додавання (486+)
ADC r/m, r/m/i	Додавання двох операндів з урахуванням переносу від попередньої операції: r/m ← (r/m + r/m/i + CF)
INC r/m	Збільшення на 1: r/m ← (r/m + 1)
SUB r/m, r/m/i	Віднімання: r/m ← (r/m - r/m/i)
SBB r/m, r/m/i	Віднімання з позичкою: r/m ← (r/m – r/m/i – CF)
DEC r/m	Зменшення на 1: r/m ← (r/m – 1)
CMP r/m, r/m/i	Порівняння – віднімання без збереження результату (тільки установка прапорів)
CMPXCHG r/m, reg	Порівняння і обмін даними (486+)
CMPXCHG8B	Порівняння і обмін 8 байт (PENTIUM+)
NEG r/m	Зміна знака операнда (перетворення в додатковому коді): r/m ← (0 – r/m)
MUL r/m	Множення AL/AX/EAX на беззнакове ціле значення r/m
IMUL r/m	Множення AL/AX/EAX на ціле знакове значення r/m
IMUL reg16/32, r/m	Знакова множення reg16/32 на r/m (поміщення результату без розширення розрядності в reg16/32) (16 біт – 286+; 32 біти – 386+)
IMUL reg16/32, r/m, immed	Знакове множення r/m на 16/32-х бітовий безпосередній операнд і поміщення результату без розширення розрядності в reg16/32 (16 біт – 286+; 32 біти – 386+)
DIV r/m	Ділення розширеного акумулятора на беззнакове число з r/m
IDIV r/m	Знакове ділення розширеного акумулятора на знакове ціле з r/m
CBW	Знакове розширення байта в акумуляторі (AL) до слова: AH ← заповнюється бітом AL [7]
CWD	Перетворення слова в подвійне слово (розширення знака AX в DX) DX ← заповнюється бітом AX [15]
CWDE	EAX [16…31] ← заповнюється бітом AX [15]
CDQ	Перетворення подвійного слова у збільшене в чотири рази: EDX ← заповнюється бітом EAX [31]
DAA	Корекція AL після BCD-додавання
DAS	Корекція AL після BCD-віднімання
AAA	Корекція AL після ASCII-додавання
AAS	Корекція AL після ASCII- віднімання
AAM	Корекція AL після ASCII-множення
AAD	Корекція AL, AH перед ASCII-діленням

Операції додавання і віднімання однакові для обох типів чисел. Єдина відмінність полягає у механізмі виявлення виходу за діапазон. Команди

додавання і віднімання встановлюють прапор CF, якщо результат, що інтерпретується як беззнакове число, виявляється поза діапазоном; вони ж встановлюють прапор OF, якщо результат, що інтерпретується як знакове число, виходить за діапазон.

Команда **XADD** – обміну та додавання – обмінює операнди і додає їх. Тому на місці операнда-джерела залишається операнд-одержувач, а на місці операнда-отримувача формується сума.

Команда **NEG** змінює знак операнда в додатковому коді.

Команда **CMP** (порівняння) аналогічна команді віднімання, але результат ніде не запам'ятовується. Ця команда виставляє прапори, за якими можна визначити відношення між двома операндами: рівність, більше або менше (див. табл. 6). Після команди **CMP** зазвичай використовується команда умовного переходу.

Команда **CMPXCHG** – порівняння і обміну – сприймає 3 операнди: операнд-джерело в регістрі, операнд-одержувач у пам'яті і акумулятор AL/AX/EAX. Якщо значення в операнді-одержувачі та акумуляторі рівні, операнд-одержувач замінюється операндом-джерелом. Інакше початкове значення операнда-одержувача завантажується в акумулятор. Прапори відображають результат, отриманий при відніманні операнда-одержувача з акумулятора.

Таблиця 2.6 – Стан прапорів після команди порівняння

Відношення	Знакові числа	Беззнакові числа
(dest)> (src)	(ZF=0) & (SF=OF)	(CF=0) & (ZF=0)
(dest) => (src)	SF = OF	CF = 0
(dest) = (src)	ZF = 1	ZF = 1
(dest) <= (src)	(ZF=1) & (SF<>OF)	(CF=1) & (ZF=1)
(dest) < (src)	SF <> OF	CF = 1

Команди множення можуть мати: одно-, дво- або триадресну форму.

У одноадресних командах **MUL** і **IMUL** один із співмножників за замовчуванням розміщується в акумуляторі (див. табл. 7), а другий співмножник вказаний в команді. Результат множення вдвічі довший за операнди.

Таблиця 2.7 – Розміщення першого множника і результату множення

Розрядність операндів	Множник	Результат	
		Старша частина	Молодша частина
8	AL	AH	AL
16	AX	DX	AX
32	EAX	EDX	EAX

При двоадресній формі **(IMUL reg16/32,r/m)** або триадресній формі **(IMUL reg16/32, r/m, immed)** команд множення зі знаком – результат розміщується в регістрі-приймачі. У цьому випадку старші 16 (або 32) розряди добутку при множенні 16-ти (або 32-х) розрядних операндів втрачаються. Такі команди зручно застосовувати для обчислення адрес елементів масивів.

Команди ділення **DIV** і **IDIV** мають тільки одноадресну форму, причому розрядність діленого (див. табл. 8) повинна вдвічі перевищувати розрядність дільника, зазначеного в команді.

Знак залишку при виконанні команди **IDIV** встановлюється рівним знаку діленого.

Таблиця 2.8 – Розміщення діленого і результатів ділення

Розрядність дільника	Ділене		Частка	Залишок
	Старші розряди	Молодші розряди		
8	AH	AL	AL	AH
16	DX	AX	AX	DX
32	EDX	EAX	EAX	EDX

Для підготовки операндів-діленого подвійної довжини використовуються команди розширення акумулятора знаковими бітами. При виконанні команд – **CBW / CWDE** (перетворення байта в слово / перетворення слова в подвійне слово з розширенням в акумуляторі) – розширений операнд залишається в акумуляторі. Команди – **CWD / CDQ** (перетворення слова в подвійне слово / перетворення подвійного слова в зчетверене слово) – розширюють акумулятор AX або EAX до регістрів DX або EDX відповідно, куди заноситься старша половина (розширений знак) операнда.

Система команд процесорів x86 дозволяє виконувати арифметичні дії над числами, поданими у ***двійково-десятковому упакованому форматі*** (**BCD**-код) або у коді **ASCII**, що використовується при обміні інформацією і при введенні з клавіатури. Для цих чисел допустимі значення від 0 до 9 в молодшій тетраді.

Команда **DAA** – ДЕСЯТКОВОЇ ***корекції акумулятора після додавання*** BCD-чисел виконує дії над вмістом AL наступним чином:

– якщо вміст молодшої тетради AL більший за 9 чи встановлений прапор AF = 1, то до вмісту AL додається 6;

– якщо після цього вміст старшої тетради AL став більшим за 9 чи встановлений прапор CF, то число 6 додається до старшої тетради AL.

Аналогічно виконуються дії над вмістом AL командою **DAS** – ***десяткова корекція після віднімання*** BCD-чисел:

– якщо молодша тетрада більша за 9 чи встановлений прапор AF = 1, то з AL віднімається число 6;

– якщо після цього старша тетрада більша за 9 чи встановлений прапор CF = 1, то число 6 віднімається із старшої тетради AL.

Перед виконанням арифметичних команд над числами в коді ASCII необхідно очистити старші тетради цих чисел. Такі числа називаються: розпакованими (незапакованими).

Команда **AAA** виконує корекцію числа в регістрі AL, отриманого в результаті додавання двох розпакованих десяткових операндів. Якщо вміст молодшої тетради AL більший за 9 чи встановлений прапор AF = 1, то до вмісту AL додається 6; після цього до AH додається 1, очищається старша тетрада AL, і встановлюються прапори CF і AF.

Команда **AAS** виконує корекцію числа в регістрі AL, отриманого в результаті віднімання двох розпакованих десяткових операндів. Якщо вміст молодшої тетради AL більший за 9 чи встановлений прапор AF = 1, то з AL віднімається число 6; після цього з AH віднімається 1, очищається старша тетрада AL, і встановлюються прапори CF і AF.

Команда **AAM** виконує корекцію числа в регістрі AL, отриманого після множення двох розпакованих десяткових операндів. Вміст AL ділиться на 10; частка пересилається в AH, а залишок – в AL.

Команда **AAD** проводить корекцію діленого до виконання команди ділення. Для цього вміст регістра AH множиться на 10 і результат додається до вмісту AL, старший байт акумулятора AH очищається. Отриманий операнд використовується для звичайного ділення на розпакований дільник.

Логічні двооперандні команди служать для реалізації трьох булевих функцій (результат поміщається на місце першого операнда):

– AND – порозрядне логічне І;

– OR – порозрядне логічне АБО;

– XOR – порозрядне логічне ВИКЛЮЧАЄ АБО (сума по модулю 2).

Сюди також належить команда TEST (перевірка), що виконує порозрядне логічне І, але результат нікуди не заносить, а тільки встановлюються прапори для виконання умовних переходів.

Команди XOR і SUB дозволяють обнулити всі біти регістра (регістр має бути і джерелом і приймачем).

Таблиця 2.9 – Команди логічних операцій

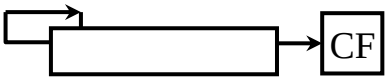
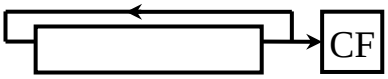
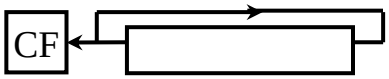
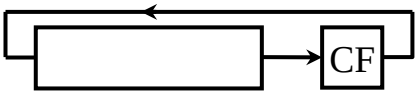
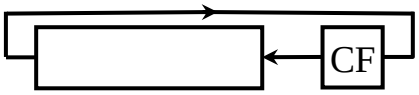
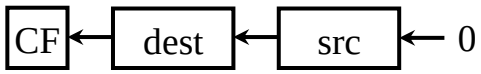
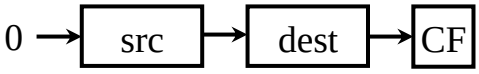
AND r/m, r/m/i	Побітове логічне І
TEST r/m, r/m/i	Перевірка біт (логічне І без запису результату – установка прапорів)
OR r/m, r/m/i	Побітове логічне АБО
XOR r/m, r/m/i	Побітове логічне ВИКЛЮЧАЮЧЕ АБО
NOT r/m	Побітова інверсія

Команди зміщень та циклічних зміщень (табл. 2.10) виконують зміщення 8/16/32-х бітового операнда на 1 біт або на довільне число біт (але не більше довжини операнда). Для зміщень більше, ніж на один біт, число зміщень може бути записане попередньо в регістр CL або задане безпосереднім операндом у команді (286+). У всіх командах зміщень останній біт, що висувається поміщається в прапор CF.

У командах подвійного зміщення операндом-приймачем (dest) може бути вміст reg16/32 або mem16/32, операндом-джерелом (src) – тільки вміст регістра загального призначення (з розрядністю 16/32). Для зміщень більших, ніж на один біт, число зміщень може бути записаним попередньо в регістр CL або заданим безпосереднім операндом у команді.

Всередині процесора операнди dest і src об'єднуються у проміжному регістрі подвійної довжини, вміст якого логічно зміщується вліво або вправо. Після зміщення в операнд-приймач (dest) поміщаються відповідні зміщені біти проміжного регістра. Вміст операнда-джерела (src) не змінюється. Можна сказати, що в цих командах зміщується операнд-приймач (dest) і в його біти, що звільняються «вставляється» вміст операнда-джерела (src).

Таблиця 2.10 – Команди зміщень

Команда	Мнемоніка	Виконання команди
Логічне зміщення вліво Арифметичне зміщення вліво	SHL SAL	
Логічне зміщення вправо	SHR	
Арифметичне зміщення вправо	SAR	
Циклічне зміщення вправо	ROR	
Циклічне зміщення вліво	ROL	
Циклічне зміщення вправо через прапор CF	RCR	
Циклічне зміщення вліво через прапор CF	RCL	
Подвійне зміщення вліво (386+)	SHLD	
Подвійне зміщення вправо (386+)	SHRD	

Команди бітових операцій відсутні у мікропроцесорів 86/286.

Команда **BT r / m, im8** або **BT r / m, reg** (тестування біта) вибирає з адресованого регістра або пам'яті (r/m) значення певного біта і копіює його у прапор CF. Номер біта (індекс) визначається значенням байта безпосереднього операнда або задається вмістом регістра (reg).

Коли номер біта (індекс) визначено як константа (immed), його діапазон становить від 0 до 31. Якщо поле r/m визначає комірку пам'яті (розміром слово або подвійне слово), а номер біта заданий вмістом регістра reg, то цей номер

біта (індекс) вважається цілим знаковим числом в діапазоні від –32К до + (32К–1) для 16-ти бітової операції або від –2Г до + (2 Г – 1) для 32-х бітової операції.

Аналогічна команда **BTS** після копіювання встановлює адресований біт в 1. Команда **BTR** після копіювання скидає біт, а команда **BTC** – інвертує.

Команди **BSF** і **BSR** проводять сканування вмісту регістра або комірки пам'яті (r/m) і заносять в регістр-приймач (reg) номер першого одиничного біта, що зустрівся. При виконанні команди BSF сканування починається з молодшого розряду, а в команді BSR зі старшого розряду. Якщо операнд дорівнює нулю (одиничні біти відсутні), то встановлюється прапор ZF = 1. При цьому вміст регістра-приймача буде невизначеним. Якщо одиничний біт знайдений, то прапор ZF = 0.

Таблиця 2.11 – Команди бітових операцій (386+)

BT r/m, im8 BT r/m, reg	Тестування біта – завантаження в CF біта з номером (індексом) im8 з r/m. Завантаження в CF біта з r/m з номером з «reg»
BTC r/m, im8 BTC r/m, reg	Тестування (завантаження в CF) і інверсія біта
BTR r/m, im8 BTR r/m, reg	Тестування (завантаження в CF) і скидання біта
BTS r/m, im8 BTS r/m, reg	Тестування (завантаження в CF) і установка в 1 біта
BSF(BSR) reg, r/m	Сканування біт вперед (назад) в комірці r/m. В reg завантажується індекс першого одиничного біта в комірку r / m.

Команди обробки ланцюжків. Під ***ланцюжком*** **(*рядком*)** розуміють послідовність байтів, слів або подвійних слів у пам'яті, а ***ланцюжковою*** **(*рядковою*)** ***операцією*** називається операція, що виконується над кожним елементом ланцюжка. Наприклад, ланцюжкова передача виробляє пересилання цілого ланцюжка з однієї області пам'яті в іншу. Скорочення часу виконання ланцюжкових команд досягається за рахунок потужного набору примітивних команд, що виконують прискорену обробку кожного елемента ланцюжка і необхідні службові дії (табл. 2.12).

Перед виконанням ланцюжкових команд необхідно:

– завантажити початкову (кінцеву) адресу ланцюжка-джерела в регістри DS: (E)SI (допускається заміна сегменту) (є відповідні команди: LDS та ін.);

– завантажити початкову (кінцеву) адресу ланцюжка-приймача в регістри ES: (E) DI (командою LES);

– скинути прапор DF=0 (командою CLD), якщо ланцюжки обробляються за зростанням адрес, або встановити прапор DF=1 (командою STD), якщо ланцюжки обробляються за спаданням адрес;

– при використанні префікса повторення REP в регістр (E)CX завантажити кількість повторень ланцюжкової операції;

– при роботі з портами в регістр DX завантажити адресу порту.

Таблиця 2.12 – Примітиви ланцюжкових (рядкових) команд

MOVS	mem(DI) ← mem(SI),	Модифікувати SI, DI
CMPS	mem(SI) – mem(DI), FLAGS,	Модифікувати SI, DI
SCAS	A – mem(DI), FLAGS,	Модифікувати DI
LODS	A ← mem(SI),	Модифікувати SI
STOS	mem(DI) ← A,	Модифікувати DI
INS	mem(DI) ← port(DX),	Модифікувати DI (286+)
OUTS	port(DX) ←mem(SI),	Модифікувати SI (286+)

Ланцюжковий примітив **MOVSB (MOVSW, MOVSD)** – передати елемент ланцюжка – пересилає байт (слово або подвійне слово) з комірки пам'яті, зміщення якої знаходиться в регістрі (E)SI (мається на увазі, що ланцюжок-джерело за замовчуванням знаходиться в поточному сегменті даних, що визначається регістром DS, але допускається заміна сегменту), в комірку пам'яті зі зміщенням з (E)DI (ланцюжок-одержувач повинен знаходитися тільки в сегменті, що визначається регістром ES).

При виконанні ланцюжкової команди вміст регістрів (E)SI і (E)DI автоматично модифікується так, щоб адресувати наступні елементи ланцюжків. Прапор DF визначає автоінкремент (DF=0) або автодекремент (DF = 1) індексних регістрів. Величина інкремента / декремента залежить від розміру елементів і становить 1, 2 або 4, коли елементами ланцюжків є, відповідно, байти, слова або подвійні слова.

Якщо в ланцюжкову команду додати префікс повторення **REP MOVS**, то примітив MOVS, буде повторюватися зі зменшенням (E)CX на 1 (після виконання примітиву) до обнулення (E)CX.

Команда порівняння ланцюжків **CMPSB (CMPSW, CMPSD)** здійснює віднімання байта (слова або подвійного слова) ланцюжка приймача (dest) з відповідного елемента ланцюжка-джерела (src). Залежно від результату віднімання встановлюються прапори (в регістрі (E)FLAGS), але самі операнди не змінюються. Індексні регістри-вказівники просуваються на наступні елементи ланцюжків.

Коли перед командою **CMPS** вказано префікс повторення **REPE** або **REPZ**), операція інтерпретується як: «порівнювати, поки не досягнуть кінця ланцюжків або поки не знайдений рівний елемент».

При наявності префікса **REPNE** (або **REPNZ**) операція набуває сенсу: «порівнювати, поки не досягнуть кінця ланцюжків або поки елементи залишаються рівними».

Команда сканування ланцюжків **SCASB (SCASW, SCASD)** здійснює віднімання елемента ланцюжка (байт, слово або подвійне слово) з вмісту акумулятора AL/AX/EAX. Залежно від результатів віднімання встановлюються прапори, але значення операндів не змінюється.

З префіксом **REPE** (або **REPZ**) команду **SCAS** можна використати для пошуку елемента ланцюжка зі значенням, що відрізняється від заданого в акумуляторі значення. Префікс **REPNE** (або **REPNZ**) дозволяє знайти елемент ланцюжка, значення якого дорівнює значенню в акумуляторі.

Команда **LODSB (LODSW, LODSD)** завантажує в акумулятор (AL/AX/EAX) елемент з ланцюжка (байт, слово або подвійне слово) і просуває покажчик (E)SI на наступний елемент. Зазвичай ця команда з префіксом повторення не використовується.

Команда збереження акумулятора в ланцюжку **STOSB (STOSW,STOSD)** передає байт (слово або подвійне слово) з акумулятора AL/AX/EAX в елемент ланцюжка і просуває регістр-вказівник (E)DI на наступний елемент. З префіксом повторення **REP** ця команда зручна для ініціалізації ланцюжка на фіксоване значення.

Команди введення і виведення ланцюжків **INSB (INSW, INSD)** і **OUTSB (OUNSW, OUNSD)** як і звичайні команди введення/виведення є привілейованими.

Команда **INS** вводить дані з порту, що адресується регістром DX, в комірку пам’яті з адресою ES: (E)DI. Після введення операнда проводиться модифікація регістра (E)DI на 1, 2 або 4 з урахуванням стану прапора напрямку DF.

Команда **OUTS** виводить дані з комірки пам’яті з адресою DS: (E)SI у вихідний порт, адреса якого знаходиться в регістрі DX. Після виведення операнда проводиться корекція вказівника (E)SI.

Обидві ці команди можуть використовуватися з префіксом повторення **REP**. У цьому випадку введення або виведення даних повторюється до обнулення регістра-лічильника (E)CX.

Необхідно відзначити, що п'ять мнемонік префікса повторення **REP, REPE / REPZ, REPNE / REPNZ** визначають тільки два об’єктних (машинних) коди префікса (0F2h і 0F3h), а п’ять мнемонік введені для кращої передачі змістовного сенсу завдання.

Команди роботи з прапорами. Однобайтові команди цієї групи дозволяють модифікувати деякі прапори регістра (E)FLAGS. Решта прапорів можуть бути модифіковані після запису вмісту регістра прапора в регістр або комірку пам’яті (наприклад, командою PUSHF (D)), з подальшим поверненням у прапоровий регістр.

Команди, що модифікують прапор IF, є IOPL-чутливими, тобто програма, що їх виконує повинна мати поточний рівень привілеїв CPL, менший або рівний вмісту поля IOPL в регістрі (E)FLAGS. Якщо ця умова не виконується, виникає порушення загального захисту.

Команди передачі управління. Команда безумовного переходу із загальною мнемонікою JMP має 5 форм, що розрізняються відстанню до адреси призначення від поточної команди і способом завдання призначення (цільової адреси target).

У короткому (SHORT) внутрішньосегментному переході двобайтова команда JMP rel8 містить у другому байті зміщення в додатковому коді (максимально можливий перехід: назад – 128 або вперед +127 від адреси команди, що знаходиться після команди JMP).

Таблиця 2.13 – Команди роботи з прапорами

CLC	CF← 0	Скидання прапора перенесення
CMC	CF← 1 – CF	Інверсія прапора перенесення
STC	CF← 1	Установка прапора перенесення
CLD	DF← 0	Скидання прапора напряму ланцюжків DF
STD	DF← 1	Установка прапора напряму DF
CLI	IF← 0	Заборона маскованих апаратних переривань
STI	IF← 1	Дозвіл маскованих апаратних переривань
CTS (CLTS)	TF ← 0	Скидання прапора переключення задач
LAHF	Завантаження молодшого байта регістра прапорів в AH	
SAHF	Збереження AH в молодшому байті регістра прапорів	

Команда прямого внутрішньосегментного переходу (NEAR) аналогічна попередній, але повне зміщення в додатковому коді містить 16 (або 32 біта), що додається до поточного значення (E)IP. Ця форма команди передає управління в будь-яку точку поточного сегмента коду.

У команді непрямого внутрішньосегментного переходу JMP r/m адреса цільового призначення (target) завантажується в (E)IP з регістра або комірки пам’яті.

Команда прямого міжсегментного переходу JMP prt містить безпосередній операнд, що містить: 16-ти бітовий селектор, який завантажується в регістр CS, і 16-ти (або 32-х) бітовий зсув, що завантажується в (E)IP.

Команда непрямого міжсегментного переходу адресує в пам’яті повний 32-х (або 48-ми) бітовий покажчик – селектор: зсув. Селектор завантажується в регістр CS, а зміщення – у регістр (E)IP.

Команди умовних переходів (табл. 2.14) здійснюють передачу управління в залежності від результатів попередніх операцій. Всі команди умовних переходів виробляють передачу управління тільки в межах поточного сегмента коду (тобто вміст сегментного регістра CS не змінюється), якщо задана в команді умова задовольняється. Перехід реалізується додаванням зсуву, що знаходиться в команді (у додатковому коді), до вмісту регістра (E)IP. У процесорах 86/286 8-ми бітовий зсув забезпечує діапазон переходу від – 128 до +127 байт. У процесорах 386+ поряд з таким зміщенням допускається також повний 16-ти або 32-х бітовий зсув у додатковому коді. Цим забезпечується перехід в будь-яку точку поточного сегмента коду.

Таблиця 2.14 – Команди передачі управління (переходів)

JMP target	Безумовний перехід до цільової адреси target
J(E)CXZ target	Умовний перехід, якщо (E)CX = 0
LOOP target	Декремент (E)CX і перехід, якщо (E)CX<> 0
LOOPE target (LOOPZ) target	Декремент (E)CX и перехід, якщо (E)CX <> 0 & ZF = 1
LOOPNE target (LOOPNZ) target	Декремент (E)CX и перехід, якщо (E)CX <> 0 & ZF = 0
Jccc target	Команди умовного переходу
CALL target	Виклик процедури (підпрограми)
RET (n)	Повернення з процедури. Необов'язковий параметр *n* задає корекцію значення вказівника стека
SETccc r/m	Умовне заповнення байта. Якщо виконується умова «ccc», усі біти байта dest (регістра або пам'яті) встановлюються в 1, інакше – в 0. Умови «ccc» ті ж, що і в командах умовних переходів (386+)

У мнемокодах команд умовних переходів при порівнянні чисел із знаком використовуються літери: **G** (greater) – більше; **L** (less) – менше.

Для чисел без знака: **A** (above) – над, вище; **B** (below) – під, нижче.

Умова рівності: **E** (equal) – рівно.

Невиконання деякої умови: **N** (not) – ні.

Для деяких команд умовних переходів зарезервовані два або три альтернативних мнемокоди (див. табл. 15), що підкреслюють змістове значення умови, яка перевіряється.

Команда виклику підпрограми (процедури) **CALL** передає управління з автоматичним збереженням в стеку адреси повернення (поточного вмісту IP), тобто адреси команди, що знаходиться після команди CALL. В кінці підпрограми остання команда RET відновлює з стека в регістр IP адресу повернення.

Команда **CALL** має такі ж форми (відносну, пряму і непряму), як і команда **JMP**; відсутня тільки коротка (SHORT) форма. За впливом на регістри CS і (E)IP команда CALL також відповідає команді JMP, але додатково включає в поточний сегмент стека адресу повернення з відповідною корекцією покажчика стека (E)SP.

Команда **RET** допускає вказівку в поле операнда безпосередньої константи immed16. В таких командах після вилучення з стека адреси повернення константа immed16 додається до вмісту регістра (E)SP. В результаті в стеці пропускаються параметри, передані підпрограмі.

Команда заповнення байта за умовою (SETcccr8 / m8) призначена для того, щоб зберегти зафіксовану прапорами умову для подальших обчислень. Мнемоніка умови «ccc» повністю збігається з умовою переходів (табл. 15).

Таблиця 2.15 – Кодування умов переходу

Код поля ссс	Мнемоніка поля ссс	Стан прапорів	Умови переходу
0000	O	OF=1	Переповнення
0001	NO	OF=0	Не переповнення
0010	B/NAE/C	CF=1	Нижче / не вище або рівно
0011	AE/NB/NC	CF=0	Не нижче / вище або рівно
0100	E/Z	ZF=1	Рівно / нуль
0101	NE/NZ	ZF=0	Не рівно / не нуль
0110	BE/NA	CF=1 & ZF=1	Нижче або рівно / не вище
0111	NBE/A	CF=0 & ZF=0	Не нижче або рівно / вище
1000	S	SF=1	Є знак (від'ємний)
1001	NS	SF=0	Нема знака (додатний)
1010	P/PE	PF=1	Є паритет / парний паритет
1011	NP/PO	PF=0	Нема паритету / непарний паритет
1100	L/NGE	ZF<>OF	Менше / не більше або рівно
1101	NL/GE	SF=OF	Не менше / більше або рівно
1110	LE/NG	(SF<>OF) & ZF=1	Менше або рівно / не більше
1111	NLE/G	SF=(OF & ZF)	Не менше або рівно / більше

Команди переривання. Двобайтова команда **INT n** (табл. 16) на початку включає в стек вміст регістра прапорів (E)FLAGS і повну адресу повернення, подану вмістом регістрів CS і (E)IP. Крім цього скидається в нуль прапор дозволу переривань IF. Після цього здійснюється непрямий перехід через елемент «n» дескрипторної таблиці переривань IDT.

Однобайтовий варіант цієї команди **INT 3** називається перериванням контрольної точки.

Команда переривання **INTO** еквівалентна команді **INT 4**, якщо встановлено прапор переповнення OF = 1. Коли ж прапор OF = 0, команда INTO не виконує ніяких дій.

Команда повернення з переривання **IRET** витягує із стека збережені в ньому адресу повернення і регістр прапорів.

Таблиця 2.16 – Команди переривання

INTn	Виконання програмного переривання
INT 3	Однобайтова команда переривання за типом 3
INTO	Виконання програмного переривання 4, якщо OF=1
IRET	Повернення з переривання

Лабораторна робота № 1 Програмування в машинних кодах

А. Арифметичні операції: метод мікропроцесора Intel

1. Запишіть в регістри AX і BX числа 3A7h і 1EDh за допомогою команди R:

```
-r ax
AX 0000
:3A7
-
-r bx
BX 0000
:92A
-r
AX=03A7  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=14AB  ES=14AB  SS=14AB  CS=14AB  IP=0100   NV UP EI PL NZ NA PO NC
14AB:0100 1900      SBB     [BX+SI],AX                  DS:092A=0000
-
```

Починаючи з адреси CS:100 введіть команду додавання цілих чисел:

```
-a 100
14AB:0100 add ax,bx
14AB:0102
```

Введіть команду U 100, щоб побачити результат:

```
-u100
14AB:0100 01D8          ADD     AX,BX
  ...                   ...
```

Виконайте цю команду в режимі трасування:

```
-t
```

AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000 DS=14AB ES=14AB SS=14AB CS=14AB IP=0102 NV UP EI PL NZ AC PE NC 14AB:0102 1277D7 ADC DH,[BX-29] DS:0901=00

```
-
```

Тепер регістр AX містить число CD1h, яке є сумою чисел CD1h і 92Ah.

2. Виконайте кроки п.1 для ознайомлення з роботою команди віднімання SUB.

3. Команда множення регістрів AX на BX розміщує старші 16 розрядів в регістрі DX, а молодші - в AX. Команда ділення AX на BX частку розміщує в регістрі AX, а остачу в регістрі DX. Виконайте ці операції самостійно та перевірте одержані результати.

B. Вивід символів на екран

1. Ведіть програму виводу одного символа на екран, використовуючи команду A

(assemble) редактора програм Debug:

```
MOV DL,2A       ;код символа * (2A)
MOV AH,02       ;функція = 2
INT 21          ;переривання ДОС – вивід символа
RET         ;повернення до редактора Debug
```

2. Перевірте правильність вводу за допомогою команди U (unassemble):

```
14AB:0100 B22A      MOV    DL,2A
14AB:0102 B402      MOV    AH,02
14AB:0104 CD21      INT    21
14AB:0106 C3        RET
```

3. Виконайте програму за допомогою команди G (go). На екрані повинен з’явитись символ (*) та повідомлення про нормальне завершення програми:

```
-g
*
Нормальне завершення роботи програми
-
```

4. Змініть програму в першому рядку так, щоб вона виводила інший символ (код символа повинен бути в межах 21h до FFh). Символи ASCII розташовані в таких проміжках:

21h - 2Fh – спеціальні знаки;
30h - 39h – цифри;
3Ah - 3Fh – спеціальні знаки;
40h - 5Fh – великі латинські букви ;
60h - 7Fh – малі латинські букви;
80h - 9Fh – великі російські букви;
A0h - AFh – малі букви від 'а' до 'п';
E0h - EFh – малі букви від 'р' до 'я'.

C. Вивід рядка символів

1. Введіть спочатку рядок символів, починаючи з адреси 200h; останнім символом цього рядка повинен бути $ (код 24). Введіть такі коди, використовуючи команду

E 200:
48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E 24 <Enter>

2. Подивіться на ділянку пам’яті, що містить ці дані, ввівши команду d200:

```
-d200
14AB:0200  48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E   Hello, DOS
here. 14AB:0210  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  $...
14AB:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
14AB:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Цей рядок символів можна ввести також простіше командою E 200 'Hello, DOS here.'.

2. Введіть програму виводу рядка символів на екран:

```
MOV    AH,09          ; функція = 09
MOV    DX,0200; початкова адреса рядка символів
INT    21             ; переривання ДОС - вивід рядка символів
INT    20             ; коректний вихід в ДОС
```

3. Перевірте правильність вводу, використовуючи команду U:

```
-U100
14AB:0100 B409         MOV    AH,09
14AB:0102 BA0002       MOV    DX,0200
14AB:0105 CD21         INT    21
14AB:0107 CD20         INT    20
-
```

4. Запам’ятайте або запишіть адресу кодового сегмента CS на випадок некоректної роботи програми або ваших помилок. Виконайте програму за допомогою команди G (go):

```
-g
Hello, DOS here.
Нормальне завершення роботи програми
-
```

5. Введіть починаючи з адреси 200h інше текстове повідомлення, наприклад: "Папа, мама і Я'. Перевірте правильність роботи вашої програми з цим текстом. На жаль, команда Dump вам не допоможе, так як другу половину кодової таблиці ASCII-кодів редактор Debug сприймає як спеціальні коди, які замінює крапкою.

А тому символів кирилиці ви не побачите, - лише їх коди:

```
-e 200 'Папа, мама і Я'
-d 200
14AB:0200  8F A0 AF A0 2C 20 AC A0-AC A0 20 9F 21 72 65 2E   ....,  .... .
.e.
14AB:0210  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $...............
```

6. Перемістіть блок даних (текстовий рядок довжиною 11h байт) з адреси 200h на адресу 110h. Це потрібно для того, щоб під час запису програми в кодах на диск, її довжина була меншою. Для переміщення даних використайте команду Move:

```
-m 200 l 11 110
```

В цій команді параметр l 11 вказує на довжину ділянки пам'яті = 11. Дійсно, перевіримо це за допомогою команди D:

```
-d100
14AB:0100  B4 09 BA 00 02 CD 21 CD-20 00 00 00 00 00 00 00   ......!. .......
14AB:0110  48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E   Hello, DOS here.
14AB:0120  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $...............
```

Тепер довжину програми (два рядки по 16 байт + 1)можна знайти за командою Hex:

```
-h 121 100
0221  0021  ; друге число – різниця чисел 121h і 100h, 21h=33d
-
```

Не забудьте замінити команду MOV DX,0200 на команду з новою адресою даних

MOV DX,110 та перевірити її коректну роботу. Для зручності заповніть частину пам'яті після ваших даних нулями (або FFh), використовуючи команду F (fill):

```
-f 121 l ff 0
-d 100
14D8:0100  B4 09 BA 10 01 CD 21 CD-20 00 00 00 00 00 00 00   ......!. .......
14D8:0110  48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E   Hello, DOS here.
14D8:0120  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $...............
14D8:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
14D8:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
14D8:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
14D8:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
14D8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

7. Запишіть вашу програму на диск, спочатку задавши її ім'я, наприклад: my1.com :

```
-n my1.com
-
```

Після цього треба визначити довжину програми і записати її в регістр CX. В нашому випадку в регістр CX треба записати число 21h:

```
-r cx
CX 0000
:21
```

-r cx

CX 0021

Тепер програму довжиною 33 байта можна записати на диск командою W (write):

-

-w

Запис: 00021 байт

-

Якщо одержите повідомлення про помилку (наприклад, коли довжина програми =0), треба повторити запис знову.

8. Закінчимо роботу з редактором програм Debug:

-q <Enter>

9. Після виходу в ДОС знайдемо файл my1.com і перевіримо його роботу. Для цього після запрошення ДОС достатньо ввести його ім’я. На екрані дисплея одержимо текстове повідомлення:

Hello, DOS here.

Зверніть увагу на розмір файла – він дорівнює 33 байти!

10*. Використайте програму Debug для модифікації вашої програми в двох напрямках:

1) зменшення довжини програми за рахунок нульових невикористаних байтів.

2) збільшення обсягу тексту для виводу на екран.

Лабораторна робота № 2 Обчислення значення функції

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: $Y = \frac{15*x^2-12}{4*x+5}(x = 3)$. Результат округлити до цілого та розмістити в пам'яті.

```
int main ()                                 // початок програми на  C++
{
        long    X=3;                        // змінна для аргументу
        long    REZ;                        // змінна для результату

        _asm{                               ; початок асемблерної вставки

        lea             EBX, REZ            ; завантаження адреси результатів в регістр EBX
        mov             EAX, 4              ; EAX = 4
        imul            X                   ; EAX = 4 * x
        add             EAX, 5              ; EAX = 4 * x + 5
        mov             EDI, EAX            ; пересилання знаменника  в регістр EDI
        mov             EAX, 15             ; EAX = 15
        imul            X                   ; EAX = 15 * x
        imul            X                   ; EAX = 15 * x^2
        sub             EAX, 12             ; EAX = 15 * x^2 - 12
        cdq                                 ; розширення операнда-ділимого в EAX-EDX
        div             EDI                 ; часне – EAX  , залишок – EDX
        shr             EDI, 1              ; ділення знаменника на 2
        cmp             EDI, EDX            ; порівняння половини дільника з залишком
        adc             EAX, 0              ; додавання до часного заєму від порівняння
        mov             dword ptr[EBX], EAX ; пересилання  результату в пам'ять

        }                                   // закінчення асемблерної вставки
}
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
    a. Титульний лист.
    b. Виконаний приклад.
    c. Тест виконання прикладу (порівняння результату з іншою програмою).
    d. Вихідний код індивідуального завдання.
    e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант	Функція	Значення *x*
1.	$Y = 5 * x^2 + 2 * x – 14$	3
2.	$Y =7500 / (2 * x^2 + 15)$	5
3.	$Y = (6 * x^2 + 12) / (5 * x – 8)$	7
4.	$Y = (2500 * x – 8) / (3 * x^2 + 20)$	2
5.	$Y = (8 * x^2 + 12 * x – 7) / (3 * x + 25)$	4
6.	$Y = 7 * x^2 + 12 * x – 32$	8
7.	$Y = (5 * x^2 - 2 * x – 14)/(1-x^2)$	11
8.	$Y = (3 * x^2 + 2 * x + 14)/(1+x)$	2
9.	$Y = 2300/ ((2/3) * x^2 - 15)$	8
10.	$Y = (4 * x^2-12) / (5 * x^2 + 8)$	2
11.	$Y = (230 * x^2 + 8) / (2 * x + 20)$	5
12.	$Y = (6 * x^2 -10 * x – 7) / (3 * x - 5)$	4
13.	$Y = (7 * x^2 + 12 * x – 32)/x$	2
14.	$Y = (2 * x^2 - 2 * x - 17)/(4+x^2)$	6
15.	$Y = 4300*x/ ((1/3) * x^2 - 15)$	9
16.	$Y = (11 * x^2-12) / (5 * x^2 + 8*x-2)$	4
17.	$Y = (370 * x^2 - 8) / (2 * x - 40)$	1
18.	$Y = (6 * x^2 -10 * x + 7) / (3 * x^2 - 5)$	5
19.	$Y = (4 * x^2 - 12 * x – 32)/(x^2-51)$	7
20.	$Y = (3 * x^2 + 7 * x - 14)/(1+x^2)$	6
21.	$Y = 4300/ ((7/3) * x^2 + 5)$	8
22.	$Y = (3 * x-12) / (5 * x^2 - 8)$	12
23.	$Y = (230 * x + 8) / (2 * x^2 + 20)$	4
24.	$Y = (11 * x^2 -10 * x + 7) / (3 * x^2 - 5)$	2
25.	$Y = 4300*x^2/ ((11/3) * x^2 - 15)$	6

Лабораторна робота №3 Обчислення кількох значень функцій

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: обрахувати 7 значень функції $Y = \frac{15*x^2-12}{4*x+5}(x = 3)$. Результат округлити до цілого та розмістити в пам'яті.

```
void main ()                                // початок програми на  C++
{
        long    X=3;                        // змінна для аргументу
        long    REZ[7];                     // масив змінних для результату

        _asm{                               ; початок асемблерної вставки

        lea             EBX, REZ            ; завантаження адреси результатів в регістр EBX
        mov             ECX, 7              ; рахівник кількості повторень циклу
m1:     mov             EAX, 4              ; EAX = 4
        imul            X                   ; EAX = 4 * x
        add             EAX, 5              ; EAX = 4 * x + 5
        mov             EDI, EAX            ; пересилання знаменника  в регістр EDI
        mov             EAX, 15             ; EAX = 15
        imul            X                   ; EAX = 15 * x
        imul            X                   ; EAX = 15 * x^2
        sub             EAX, 12             ; EAX = 15 * x^2 - 12
        cdq                                 ; розширення операнда-ділимого в EAX-EDX
        div             EDI                 ; частное – EAX  , остаток – EDX
        shr             EDI, 1              ; розширення операнда-ділимого в EAX-EDX
        cmp             EDI, EDX            ; порівняння половини дільника з залишком
        adc             EAX, 0              ; додавання до часного заєму від порівняння
        mov             dword ptr[EBX], EAX ; пересилання  результату в пам'ять
        add             EBX, 4              ; збільшення адреси результатів
        add             X, 3                ; збільшення аргументу
        loop            m1                  ; зациклювання по рахівнику в ECX
        }                                   // закінчення асемблерної вставки
}
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
    a. Титульний лист.
    b. Виконаний приклад.
    c. Тест виконання прикладу (порівняння результату з іншою програмою).
    d. Вихідний код індивідуального завдання.
    e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальне завдання

Варіант	Функція	Значення *x*
1	$Y = 5 * x^2 + 2 * x - 14$	Значення *x* та крок вводяться з клавіатури
2	$Y = 7500 / (2 * x^2 + 15)$	
3	$Y = (6 * x^2 + 12) / (5 * x - 8)$	
4	$Y = (2500 * x - 8) / (3 * x^2 + 20)$	
5	$Y = (8 * x^2 + 12 * x - 7) / (3 * x + 25)$	
6	$Y = 7 * x^2 + 12 * x - 32$	
7	$Y = (5 * x^2 - 2 * x - 14)/(1-x^2)$	
8	$Y = (3 * x^2 + 2 * x + 14)/(1+x)$	
9	$Y = 2300/ ((2/3) * x^2 - 15)$	
10	$Y = (4 * x^2-12) / (5 * x^2 + 8)$	
11	$Y = (230 * x^2 + 8) / (2 * x + 20)$	
12	$Y = (6 * x^2 -10 * x - 7) / (3 * x - 5)$	
13	$Y = (7 * x^2 + 12 * x - 32)/x$	
14	$Y = (2 * x^2 - 2 * x - 17)/(4+x^2)$	
15	$Y = 4300*x/ ((1/3) * x^2 - 15)$	
16	$Y = (11 * x^2-12) / (5 * x^2 + 8*x-2)$	
17	$Y = (370 * x^2 - 8) / (2 * x - 40)$	
18	$Y = (6 * x^2 -10 * x + 7) / (3 * x^2 - 5)$	
19	$Y = (4 * x^2 - 12 * x - 32)/(x^2-51)$	
20	$Y = (3 * x^2 + 7 * x - 14)/(1+x^2)$	
21	$Y = 4300/ ((7/3) * x^2 + 5)$	
22	$Y = (3 * x-12) / (5 * x^2 - 8)$	
23	$Y = (230 * x + 8) / (2 * x^2 + 20)$	
24	$Y = (11 * x^2 -10 * x + 7) / (3 * x^2 - 5)$	
25	$Y = 4300*x^2/ ((11/3) * x^2 - 15)$	

Лабораторна робота № 4 Організація умовних переходів

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: визначити номер (n) елемента прогресії: $a_n = 8^n - 5 * n$, при якому сума елементів прогресії перевищить 10000.

```
void main ()                    // початок програми мовою с++
{
long N=0;                  // змінна пам'яті для аргументу
     long  S=0;                 //змінна для зберігання суми
     long  P=1;                 // змінна для нагромадження 8^n

     _asm{                 ; початок асемблерної вставки

m1:  inc          N              ; збільшення аргументу
     mov          EAX, 8         ; EAX = 8
     mul          P              ; множення  -  8^n
     mov          P, EAX         ; пересилання 8^n у комірку пам'яті
     add          S, EAX         ; нагромадження суми
     mov          EAX, 5         ; EAX = 5
     mul          N              ; EAX = 5 * n
     sub          S, EAX         ; нагромадження суми
     cmp          S, 10000       ; порівняння суми з 10000
     jc           m1             ; перехід, якщо сума менше 10000
     }                           // закінчення асемблерної вставки
}
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою).
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант 1. Знайти ціле значення аргументу, при якому функція $Y = 20000 / (8 * x^2 + 25)$ стане менше 20.

Варіант 2. Визначити номер (n) елемента прогресії: $a_n = n^2 + 6 * n + 28$, при якому сума елементів прогресії перевищить 1000.

Варіант 3. Знайти ціле значення аргументу, при якому функція $Y = 15 * x^2 + 11 * x - 16$ стане більше 2000.

Варіант 4. Знайти ціле значення аргументу, при якому функція $Y = (7^x) / (5 * x^2)$ перевищить 300.

Варіант 5. Знайти ціле значення аргументу, при якому функція $Y = (2000 + x) / (8 * x^2 + 25)$ стане менше 10.

Варіант 6. Знайти ціле значення аргументу, при якому функція $Y = 9 * x^2 - 8 * x + 15$ стане більше 1000.

Варіант 7. Визначити номер (n) елемента прогресії: $a_n = 5^n + 8 * n$, при якому сума елементів прогресії перевищить 20000.

Варіант 8. Знайти ціле значення аргументу, при якому функція $Y = 7 * x^2 + 25 * x - 27$ стане більше 3000.

Варіант 9. Знайти ціле значення аргументу, при якому функція $Y = 300 * x / (8^x + 14)$ стане менше 5.

Варіант 10. Визначити номер (n) елемента прогресії: $a_n = 3* n^2 - 5 * n + 12$, при якому сума елементів прогресії перевищить 1500.

Варіант 11. Знайти ціле значення аргументу, при якому функція $Y = 40000 / (8 *x^3 + 25)$ стане менше 14.

Варіант 12. Визначити номер (n) елемента прогресії: $a_n = n^3 + 4 * n^2 + 28*n+1$, при якому сума елементів прогресії перевищить 2000.

Варіант 13. Знайти ціле значення аргументу, при якому функція $Y = 5 * x^3 + 11 * x - 16$ стане більше 4000.

Варіант 14. Знайти ціле значення аргументу, при якому функція $Y = (4^x) / (5 * x^2 - 4)$ перевищить 300.

Варіант 15. Знайти ціле значення аргументу, при якому функція $Y = (2000 + x) / (8 * x^2 + 25x+2)$ стане менше 10.

Варіант 16. Знайти ціле значення аргументу, при якому функція $Y =(9 * x^2 - 8 * x + 15)/(x-1)$ стане більше 1000.

Варіант 17. Визначити номер (n) елемента прогресії: $a_n = 4^n + 8 * n+10$, при якому сума елементів прогресії перевищить 20000.

Варіант 18. Знайти ціле значення аргументу, при якому функція $Y = (7 * x^2 + 25 * x - 27)/(x+1)$ стане більше 3000.

Варіант 19. Знайти ціле значення аргументу, при якому функція $Y = (300 * x+10) / (7^x + 14)$ стане менше 5.

Варіант 20. Визначити номер (n) елемента прогресії: $a_n = (3* n^2 - 2 * n + 12)/ (n+1)$, при якому сума елементів прогресії перевищить 1500.

Варіант 21. Знайти ціле значення аргументу, при якому функція $Y = 10000 / (4 * x^2 - 15)$ стане менше 70.

Варіант 22. Визначити номер (n) елемента прогресії: $a_n = (n^2 + 6 * n + 28)/ (n+1)$, при якому сума елементів прогресії перевищить 800.

Варіант 23. Знайти ціле значення аргументу, при якому функція $Y = 15 * x^2 + 11 * x - 16$ стане більше 2000.

Варіант 24. Знайти ціле значення аргументу, при якому функція $Y = (5^x+20) / (5 * x^2-15)$ перевищить 300.

Варіант 25. Знайти ціле значення аргументу, при якому функція $Y = (2000 + x) / (4 * x^2 - 10)$ стане менше 2.

Лабораторна робота № 5 Організація циклів

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: упорядкувати масив методом бульбашкового сортування.

```
void main () {

long x[8]={7, 23, 56, 33, 84, 15, 11, 74};
_asm {
mov       EDX, 7            ; лічильник зовнішнього циклу - на 1 менше
; кількості елементів масиву
m3:  lea         EBX, x            ; початкова адреса масиву
mov       ECX, EDX          ; лічильник внутрішнього циклу
m2:  mov         EAX, dword ptr[EBX]     ; елемент масиву – в EAX
add       EBX, 4
cmp       EAX, dword ptr[EBX]     ; порівняння сусідніх елементів
jc        m1                      ; перехід, якщо менше
xchg      dword ptr[EBX], EAX     ;     обмін елементів масиву
mov       dword ptr[EBX-4], EAX   ; обмін елементів масиву
m1:  loop        m2                      ; закінчення внутрішнього циклу
dec       EDX               ; зменшення лічильника зовнішнього циклу
jnz       m3                ; закінчення зовнішнього циклу
}}
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою)
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант 1 У пам’яті заданий масив з 25-ти елементів. Зберегти в регістрі ESI кількість парних від’ємних елементів.

Варіант 2. У пам’яті і заданий масив з 10-ти елементів. Зберегти в регістрі ESI кількість від’ємних елементів.

Варіант 3. Розрахувати й зберегти в пам’яті елементи масиву, задані функцією: $Y = n!$ (для n від 1 до 8).

Варіант 4. У пам’яті заданий масив з 10-ти елементів. Замінити ці числа добутком їх старшого й молодшого слова.

Варіант 5. У пам’яті заданий масив з 8-мі елементів. Помістити в регістр EAX максимальний елемент масиву, а в регістр ESI його адреса в пам’яті.

Варіант 6. У пам’яті заданий масив з 9-ти елементів. Відсортувати елементи масиву по зростанню.

Варіант 7. У пам’яті заданий масив з 10-ти елементів. Зберегти в регістрі ESI кількість непарних елементів.

Варіант 8. У пам’яті заданий масив з 12-ти елементів. Зберегти в регістрі EAX середнє арифметичне цих елементів. Результат округлити до цілого.

Варіант 9. У пам’яті заданий масив з 10-ти елементів. Зберегти в регістрі ESI кількість одиничних бітів у всіх елементах.

Варіант 10. У пам’яті заданий масив з 11-ти елементів. Відсортувати елементи масиву по убуванню.

Варіант 11 У пам’яті заданий масив з 15-ти елементів. Зберегти в новому масиві всі непарні елементи.

Варіант 12. У пам’яті заданий масив з 16-ти елементів. Зберегти в новому масиві всі парні елементи.

Варіант 13. Розрахувати й зберегти в пам’яті елементи масиву, задані функцією: $Y = n!/n$ (для n від 1 до 10)

Варіант 14. У пам’яті заданий масив з 20-ти елементів. Замінити ці числа сумою їх старшого й молодшого слова.

Варіант 15. У пам’яті заданий масив з 8-мі елементів. Помістити в регістр EAX максимальний парний елемент масиву, а в регістр ESI його адреса в пам’яті.

Варіант 16. У пам’яті заданий масив з 9-ти елементів. Відсортувати елементи масиву по зростанню методом вставки.

Варіант 17. У пам’яті заданий масив з 10-ти елементів. Зберегти в пам’яті тири найбільших непарних елементів.

Варіант 18. У пам’яті заданий масив з 11-ти елементів. Зберегти в регістрі EAX середнє арифметичне непарних елементів. Результат округлити до цілого.

Варіант 19. У пам’яті заданий масив з 10-ти елементів. Зберегти в регістрі ESI кількість нульових бітів у всіх елементах.

Варіант 20. У пам’яті заданий масив з 11-ти елементів. Відсортувати елементи масиву по убуванню методом вибору.

Варіант 21 У пам’яті заданий масив з 22-ти елементів. Зберегти в новому масиві всі елементи які кратні 3.

Варіант 22. У пам’яті заданий масив з 16-ти елементів. Зберегти в новому масиві всі елементи які діляться на 5 націло.

Варіант 23. У пам’яті заданий масив з 45-ти елементів. Знайти кількість елементів які діляться на 9 націло.

Варіант 24. У пам’яті заданий масив з 20-мі елементів. Помістити в регістр EAX максимальний непарний елемент масиву, а в регістр ESI його адреса в пам’яті.

Варіант 25. У пам’яті заданий масив з 12-ти елементів. Зберегти в регістрі EAX середнє квадратичне цих елементів. Результат округлити до цілого.

ЧАСТИНА ІІІ ПРОГРАМУВАННЯ МАТЕМАТИЧНОГО СПІВПРОЦЕСОРА (FPU)

3.1 Формати чисел з плаваючою точкою

Співпроцесор x87 розпізнає три формати чисел з плаваючою точкою, що зберігаються в пам'яті (рис. 3.1). Усередині співпроцесора всі числа перетворюються в розширений формат.

Кожен формат складається з трьох полів: знак (S), порядок і мантиса (рис. 5). Числа в цих форматах займають в пам'яті: 4, 8 або 10 байт.

Байт з найменшою адресою в пам'яті є молодшим байтом мантиси. Байт з найбільшою адресою містить сім старших біт порядку і ***біт знаку*** (S). Знак кодується: 0 – плюс, 1 – мінус.

У полі мантиси зберігаються тільки нормалізовані числа. Для цього необхідно скорегувати порядки (тобто зсунути кому) так, щоб в цілій частині числа (у двійковій системі числення) до коми була 1. Тому всі мантиси подаються у формі:

1.XXXXXXX...XXX , де: X = 0 або 1

Рисунок 3.1 – Формати чисел з плаваючою точкою

Але якщо старший біт завжди містить 1, мантису можна не зберігати в кожному числі з плаваючою точкою. Тому заради додаткового біта точності співпроцесор x87 зберігає числа одинарної і подвійної точності ***без старшого біта мантиси*** (з неявним старшим бітом). Винятком є кодування нуля – нульові поля мантиси і порядку.

Числа з розширеною точністю зберігаються і обробляються з ***явним старшим бітом***.

Поле порядку визначає степінь числа 2, на який потрібно помножити нормалізовану мантису для отримання початкового значення числа з плаваючою точкою.

Щоб зберігати ***від'ємні порядки***, в полі порядку знаходиться сума справжнього порядку і додатної константи, що називається зміщенням. Для одинарної точності зміщення дорівнює 127, для подвійної точності 1023, для розширеної точності 16383 (тобто половині максимального порядку). Двійковий код зміщення для всіх порядків дорівнює: 0111 ... 111.

Числа у полі порядку: 00000..00 і 11111..111 – зарезервовані для спецкодування або обробки помилок.

Запис чисел з плаваючою точкою в пам'яті:

– одинарна точність: $(-1)^S (1 . X1\ X2 ... X23) \cdot 2^{(E-127)}$;

– подвійна точність: $(-1)^S (1 . X1\ X2 ... X52) \cdot 2^{(E-1023)}$;

– розширена точність: $(-1)^S (X1. X2 ... X64) \cdot 2^{(E-16383)}$,

де S – значення знакового біта;

X1 X2 .. – біти, збережені в поле мантиси;

E – число, що зберігається в поле порядку.

Розширений формат використовується переважно всередині співпроцесора для подання проміжних результатів, щоб спростити отримання округлених остаточних результатів у форматі подвійної точності.

Таблиця 3.1 – Діапазон подання чисел у десятковій системі

Формат	Значущих десяткових цифр	Найменший степінь числа 10	Найбільший степінь числа 10
Одинарний	7	–37	38
Подвійний	15	–307	308
Розширений	19	–4931	4932

Діапазон подання чисел з плаваючою точкою в десятковій системі числення наведено в табл. 17 і на рис. 6.

Якщо результат арифметичної операції менший найменшого від'ємного числа або більший найбільшого додатного числа конкретного формату, то кажуть, що операція викликала ***переповнення***. Якщо ж результат арифметичної операції ненульовий, але знаходиться між найбільшим від'ємним і найменшим додатним числами конкретного формату, то кажуть, що в операції виникло ***антипереповнення***.

Відзначимо, що рис. 3.2 абсолютно симетричний для додатних і від'ємних чисел. Отже, операція знаходження абсолютного значення ніколи не може викликати ні переповнення, ні антипереповнення.

Співпроцесор x87 має команди, що перетворюють цілі числа в числа з плаваючою точкою і навпаки. Це необхідно при обчисленнях, в яких є числа обох форматів. Допустимі формати цілих чисел наведені на рис. 3.3:

Ціле слово – це 16-ти бітове ціле число процесора x86 у додатковому коді. ***Коротке ціле*** і ***довге ціле*** схожі на ціле слово, але їх довжина більша.

Упаковане десяткове число складається з 18 десяткових цифр, розміщених по дві в байті. Знаковий біт знаходиться в додатковому 10-му байті. Молодші сім біт цього байта мають бути нульовими.

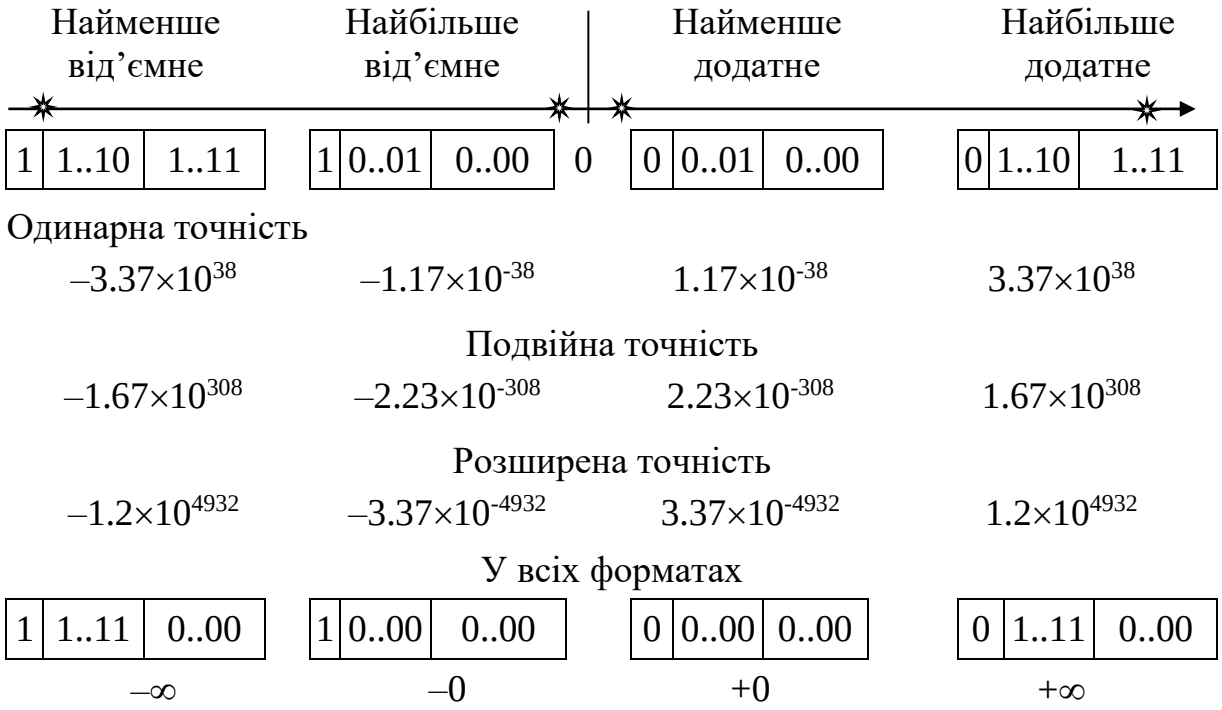


Рисунок 3.2 – Діапазон подання чисел

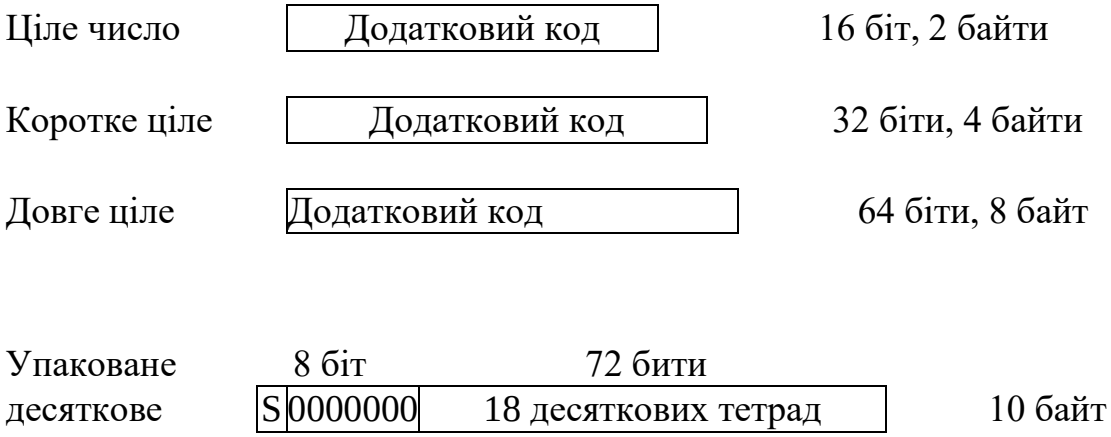


Рисунок 3.3 – Формати цілих чисел

При виконанні арифметичних операцій над числами з плаваючою точкою іноді виникають помилкові умови або ***особливі випадки***:

– ***недійсна операція*** включає в себе, наприклад, ділення і множення з операндами нескінченність і нуль, добування кореня квадратного з від'ємного числа, спробу використовувати неіснуючий регістр співпроцесора x87 та ін.;

– ***денормалізованний операнд*** виникає, коли заради збільшення діапазону доводиться жертвувати точністю;

– ***ділення на нуль*** дає в результаті нескінченність. Наявність у співпроцесора x87 двох нулів очевидним чином призводить до знакових нескінченостей:

$$x / (+0) = +\infty, \qquad -x / (+0) = -\infty,$$
$$x / (-0) = -\infty, \qquad -x / (-0) = +\infty.$$

Співпроцесори 87/287 мають два режими управління нескінченністю: проективний і афінний.

У ***проективному режимі*** факт наявності двох нескінченостей прихований (як прихований факт наявності двох нулів), тобто додатна нескінченність замикається на від'ємну нескінченність (порівняння двох нескінченостей завжди дає відповідь «рівні»). Режим проективної нескінченності зручний для обчислення раціональних функцій (значення в полюсах можна уявити як нескінченність) і ланцюгових дробів.

В ***афінному режимі*** розпізнаються дві нескінченності і два нулі; тут усі скінчені числа «*x*» задовольняють умові:

$$-\infty < x < +\infty.$$

Афінний режим ліберальніше проективного, оскільки допускає більше операцій над нескінченостями.

У співпроцесора 387+ (і 287 XL) залишено тільки афінне уявлення нескінченності.

Рисунок 3.4 – Режими округлення неточного результату

Інші особливі випадки:

– ***чисельне переповнення*** виникає, коли результат занадто великий за абсолютною величиною, щоб бути поданим у форматі приймача.

– ***чисельне антипереповнення*** виникає, коли ненульовий результат за абсолютною величиною занадто малий для подання, тобто коли він занадто близький до нуля.

– ***неточний результат*** виникає, коли результат операції неможливо точно подати в форматі приймача. Наприклад, при діленні 1.0 на 3.0 одержують нескінченний дріб, який неможливо точно подати в жодному форматі. Якщо особливий випадок неточного результату замаскований, співпроцесор x87 округлює результат до звичайного числа з плаваючою точкою. Програміст

може вибирати один з чотирьох режимів округлення (результати операцій на рис. 3.4 подані зірочками).

3.2 Регістри співпроцесора x87

Співпроцесор x87 має регістри (рис. 3.5), зручні для обчислень над числами з плаваючою точкою.

Операнди команд співпроцесора x87 можуть перебувати в пам'яті або в одному з восьми численних регістрів. Ці регістри зберігають числа переважно в форматі розширеної точності. Звернення до операндів у регістрах здійснюється набагато швидше, ніж до операндів у пам'яті.

Номер чисельного регістра, зазначеного в команді, співпроцесор завжди підсумовує з вмістом поля **TOP** (або **ST** – вершина стека) в регістрі стану. Сума (береться за модулем 8) визначає чисельний регістр, що використано, тобто регістри адресуються всередині співпроцесора, як замкнутий в кільце стек (рис. 9).

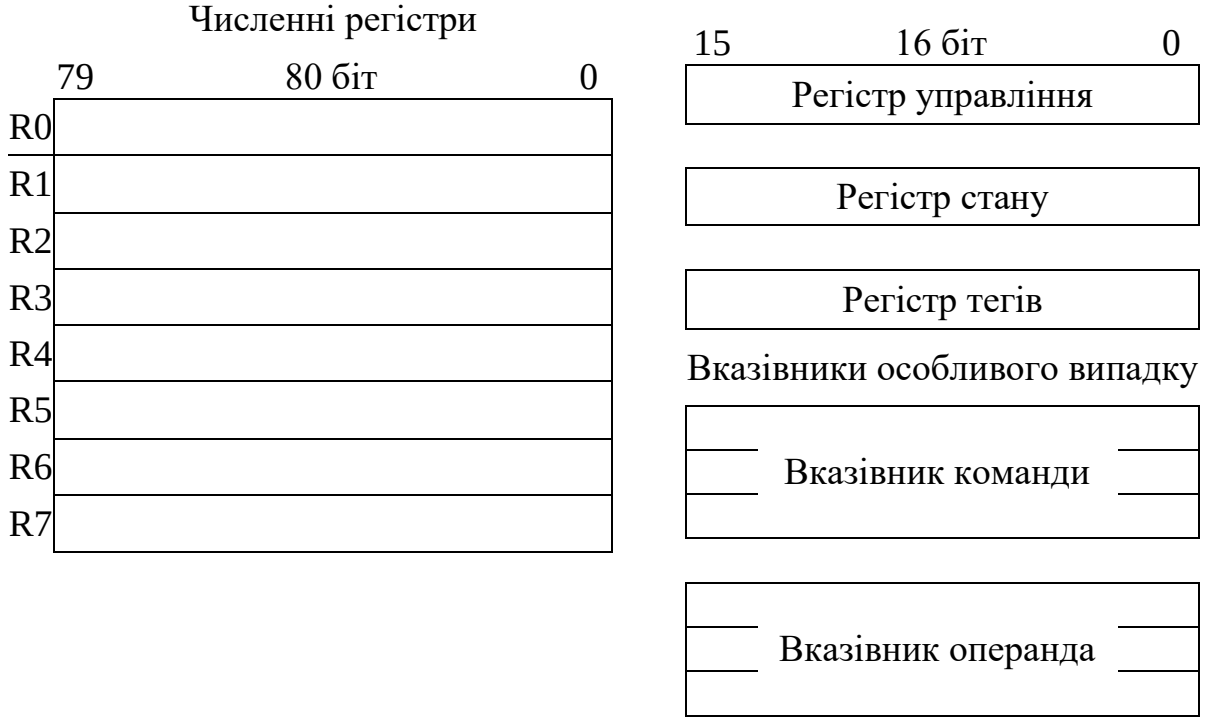


Рисунок 3.5 – Регістри співпроцесора x87

16-ти бітовий ***регістр управління*** містить поле ***масок особливих випадків*** і поля округлення чисел (рис. 3.8).

Молодші 6 бітів відведені для масок особливих випадків:

– IM – маска недійсної операції;

– DM – маска денормалізованного операнда;

– ZM – маска ділення на нуль;

– OM – маска переповнення;
– UM – маска антипереповнення;
– PM – маска точності (неточного результату).

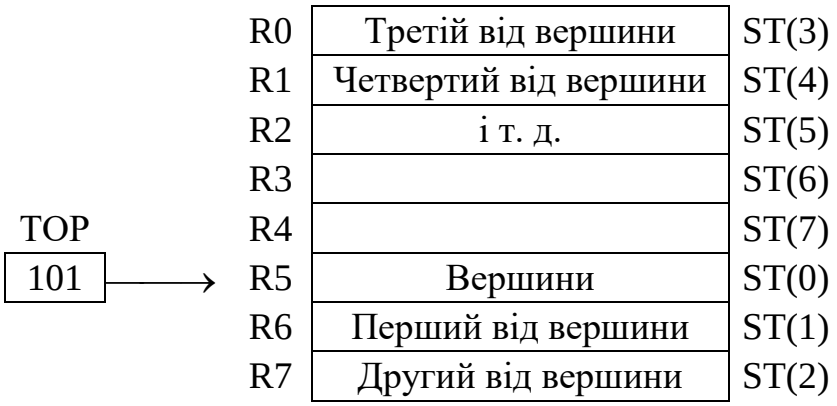


Рисунок 3.6 – Адресація численних регістрів як стека

16-ти бітовий ***регістр стану*** містить прапори, що модифікуються після виконання команд, а також поле вершини стека (TOP) (рис. 3.7).

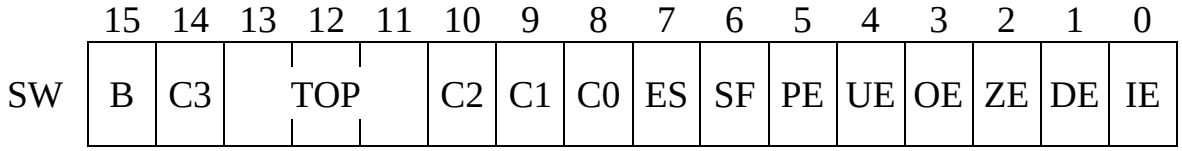


Рисунок 3.7 – Регістр стану (Status Word)

Молодші 6 біт містять ***прапори особливих випадків***:
– IE – недійсна операція;
– DE – денормалізованний операнд,
– ZE – ділення на нуль,
– OE – переповнення,
– UE – антипереповнення,
– PE – точність (неточний результат).

При виникненні чисельного особливого випадку (замаскованого чи ні) співпроцесор встановлює відповідний прапор в 1.

Прапори особливих випадків «зависають», тобто скинути їх в нуль повинен програміст, завантажуючи в регістр стану нове значення.

Біт ***порушення стека*** SF встановлюється в 1, якщо команда викликає переповнення стека (включення в уже заповнений стек) або антипереповнення стека (виключення з пустого стека). Коли SF = 1, біт коду умови C1 показує переповнення (C1 = 1) або антипереповнення (C1 = 0) стека.

Біт ***сумарної помилки*** ES встановлюється в 1, коли команда породжує будь-який незамаскований особливий випадок.

Біти C0, C1, C2, C3 містять ***коди умов***, що є результатом порівняння або команди знаходження залишку. Інтерпретація коду умови залежить від конкретної команди.

Поле ***вершини стека*** TOP (Stack Top) містить номер регістра, що є верхнім елементом стека. Його вміст додається (по модулю 8) до всіх номерів численних регістрів.

Біт ***зайнятості*** B встановлюється в 1, коли співпроцесор 8087 виконує команду або сигналізує переривання, а коли співпроцесор вільний, цей біт скидається в 0.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CW	--	--	--	IC	RC		PC		--	--	PM	UM	OM	ZM	DM	IM

Рисунок 3.8 – Регістр управління (Control Word)

16-ти бітовий ***регістр управління*** містить маски особливих випадків. Коли біт маски дорівнює 0, виникнення відповідного особливого випадку викличе призупинення програми і переривання процесора x86. Якщо ж біт маски встановлений в 1, то відповідний особливий випадок замаскований і формуються спеціальні значення чисел.

Два біти (8-й і 9-й) поля ***управління точністю*** PC змушують співпроцесор x87 округляти всі числа перед завантаженням їх у чисельні регістри до зазначеної точності:

– PC = 11 – округлення до розширеної точності (приймається за замовчуванням);

– PC = 10 – округлення до подвійної точності;

– PC = 00 – округлення до одинарної точності.

Задання пониженої точності (скорочення довжини мантиси) ліквідує переваги формату PT (розширеної точності), але не підвищує швидкодію співпроцесора.

Два біти (10-й і 11-й) поля ***управління округленням*** RC вибирають один з чотирьох режимів округлення:

– RC = 00 – округлення до найближчого (приймається за замовчуванням);

– RC = 01 – округлення до мінус нескінченності;

– RC = 10 – округлення до плюс нескінченності;

– RC = 11 – округлення до нуля.

Біт ***управління нескінченністю*** IC задає режим інтерпретації нескінченності:

– IC = 0 – проективний режим (приймається за замовчуванням);

– IC = 1 – афінний режим.

У співпроцесора 387+ біт 12 слова управління ігнорується. Використовується тільки афінний режим.

16-ти бітовий ***регістр тегів*** складається з 8-ми двохбітових полів (рис. 13). Кожне поле відповідає своєму численному регістру та індукує стан регістра:

– 00 – дійсне число (тобто будь-яке скінчене ненульове число),

– 01 – нуль,

– 10 – недійсне число (наприклад, не число або нескінченність),
– 11 – порожній регістр.

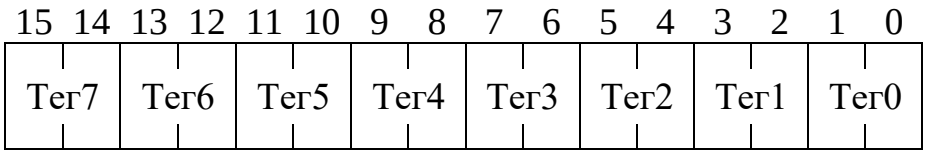


Рисунок 3.9 – Регістр тегів

Регістри відмічаються як порожні, якщо вони не ініціалізовані або їх вміст було «вилучено» зі стека численних регістрів. Співпроцесор (FPU) використовує цей тег для виявлення антипереповнення стека (занадто багато вилучень) і переповнення стека (занадто багато включень). Обидві ситуації призводять до виникнення особливого випадку недійсної операції. Коли цей особливий випадок замаскований, але в операції виникає переповнення або антипереповнення стека, співпроцесор коригує ST, як ніби нічого незвичайного не сталося, і повертає як результат операції невизначеність.

Вказівники особливого випадку (команди і операнда) (32 або 48 бітів) призначені для процедур обробки особливих випадків. Вони мають два формати в залежності від роботи співпроцесора в реальному або віртуальному режимах (рис. 3.9).

Рисунок 3.10 – Вказівники особливого випадку

У віртуальному режимі співпроцесор видає селектор і зміщення підозрілої команди і її операнда в пам’яті (якщо він є).

У реальному режимі вказівники містять 20-ти бітові адреси цих об’єктів, а також 11 молодших біт коду операції.

3.3 Команди співпроцесора x87

Формати команд співпроцесора x87 аналогічні форматам команд x86. Асемблерні мнемоніки команд співпроцесора починаються з літери **F** (Floating) і їх легко виявити, тому що таких мнемонік у процесора x86 немає.

Друга літера **I** (Integer) позначає операцію з цілим двійковим числом з пам'яті, літера **B** (Binari-codeddecimal) – операцію з десятковим операндом з пам'яті, а друга «порожня» літера визначає операцію з дійсними числами (з плаваючою точкою).

Передостання або остання літера **R** (Reveres) вказує зворотну операцію (для віднімання і ділення).

Остання літера **P** (Poping) ідентифікує команду, заключною дією якої є вилучення зі стека.

У командах неявно вказується чисельний регістр співпроцесора, що адресується вершиною стека ST або ST (0) (Stack Top).

Чисельні регістри адресуються відносно вершини стека. Наприклад, команда FADD ST(0), ST(3) додає вміст третього регістра від вершини стека до вмісту верхнього регістра стека. Верхній регістр стека можна позначити ST або ST(0). Наприклад, команда FADD ST(0), ST(0) додає вміст верхнього регістра стека до нього ж, тобто подвоює вміст регістра ST(0).

Для операндів у пам'яті допускаються всі режими адресації процесора x86, наприклад:

FADD [BX]

FADD ANAME [BX] [SI]

У командах FPU ***не використовується безпосередня адресація*** операндів.

3.3.1 Команди передачі даних співпроцесора x87. Команди включення в стек FLD, FILD, FBLD і всі команди включення констант спочатку зменшують вказівник стека на 1, перетворюють операнд-джерело в розширений формат (якщо він вже не представлений в такому форматі) і поміщають його в нову вершину стека.

Таблиця 3.2 – Команди передачі даних

Мнемоніка, операнд			Тип команди
З плаваючою точкою	Ціле число	Десяткове число	
FLD	FILD	FBLD	(–) Включити в стек
FSTP	FISTP	FBSTP	Вилучити з стека (+)
FST	FIST	-	Копіювання
FXCH	-	-	Обмін регістрів
FLDZ	(–) Включити в стек 0		
FLD1	(–) Включити в стек 1		
FLDPI	(–) Включити в стек = 3,1415...		
FLDLG2	(–) Включити в стек $\log_{10}2$		
FLDLN2	(–) Включити в стек ln2		
FLDL2T	(–) Включити в стек $\log_2 10$		
FLDL2E	(–) Включити в стек $\log_2 e$		

Мнемоніки команд з цілочисельним (двійковим) операндом починаються з «FI», а з десятковим операндом – з «FB».

Позначення:

(–) – декремент вказівника стека до включення операнда в вершину стека;

(+) – інкремент вказівника стека після вилучення операнда з вершини стека.

Команди вилучення зі стека перетворюють вміст вершини стека в необхідний формат, поміщають його в операнд-приймач (пам'ять або регістр) і після цього інкрементують вказівник стека.

Команди копіювання роблять те ж саме, але не змінюють вказівник стека. Відсутню команду FBST можна реалізувати двома командами:

FLD ST(0); продублювати вершину стека з декрементом

FBSTP; витягти з стека з інкрементом вказівника.

При виконанні команд передачі даних може виникнути необхідність вказати нову вершину стека. Команди FINCSTP і FDECSTP здійснюють відповідно інкремент і декремент вказівника стека ST. Вони не впливають на регістр тегів і чисельні регістри.

Арифметичні команди. Базові арифметичні команди додавання, віднімання множення і ділення мають два операнди (джерело і приймач) і реалізують дії:

ПРИЙМАЧ ← ПРИЙМАЧ $ ДЖЕРЕЛО,

де $ – основні команди: +, –, ×, /.

Для некомутативних команд віднімання і ділення є обернені варіанти команд (у кінці мнемоніки додається буква R – reveres):

ПРИЙМАЧ ← ДЖЕРЕЛО $ ПРИЙМАЧ.

У всіх командах один операнд має бути в вершині стека. Мнемоніки всіх базових арифметичних команд наведені в табл. 3.3.

Таблиця 3.3 – Мнемоніки базових арифметичних команд

Команди	Основна	Ціле в пам'яті	З вилученням
Додавання	FADD	FIADD	FADDP
Віднімання	FSUB	FISUB	FSUBP
Обернене віднімання	FSUBR	FISUBR	FSUBRP
Множення	FMUL	FIMUL	FMULP
Ділення	FDIV	FIDIV	FDIVP
Обернене ділення	FDIVR	FIDIVR	FDIVRP

Є 6 форм базових команд. Дії всіх шести форм команд ілюструються на прикладі команди віднімання.

FSUB mem,

FISUB mem – адресований операнд в пам'яті є джерелом, а регістр вершини стека ST(0) – приймачем. Перетворення в розширений формат з

плаваючою точкою здійснюється в процесі виконання команди. Вказівник стека не модифікується.

FSUB ST, ST(i) – будь-який чисельний регістр ST(i) є джерелом, а ST(0) – приймачем. Вказівник стека не модифікується.

FSUB ST(i), ST – вершина стека є джерелом, а ST(i) – приймачем. Вказівник стека не модифікується.

FSUBP ST(i), ST – вершина стека є джерелом, а ST(i) – приймачем. Після закінчення операції джерело ST(0) вилучають із стека з подальшим інкрементом вказівника стека (рис. 3.11).

FSUB (команда з класичною стековою адресацією – використовує тільки ST1, ST0) витягує з вершини стека джерело (потім інкрементує вказівник стека), потім витягує приймач (ще раз інкрементує покажчик стека), виконує операцію і перед включенням результату в стек декрементує вказівник. У підсумку вершина стека змістилася у бік збільшення (рис. 3.12). Остання форма є частинним випадком попередньої.

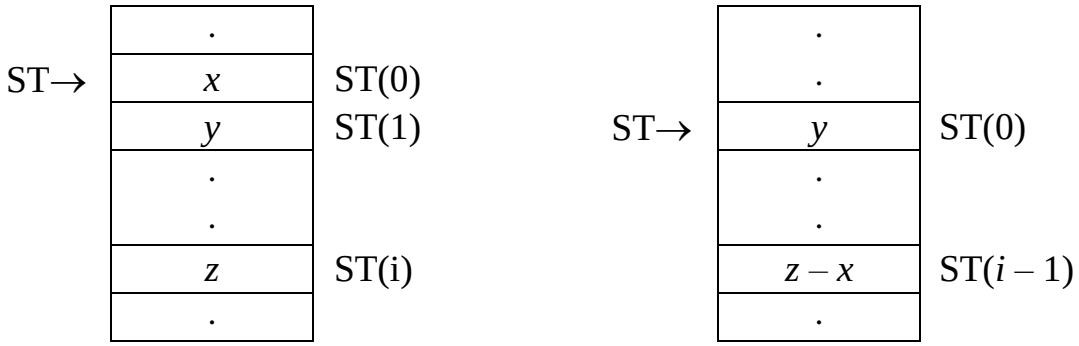


Рисунок 3.11 – Дія команди FSUBP ST(i),ST

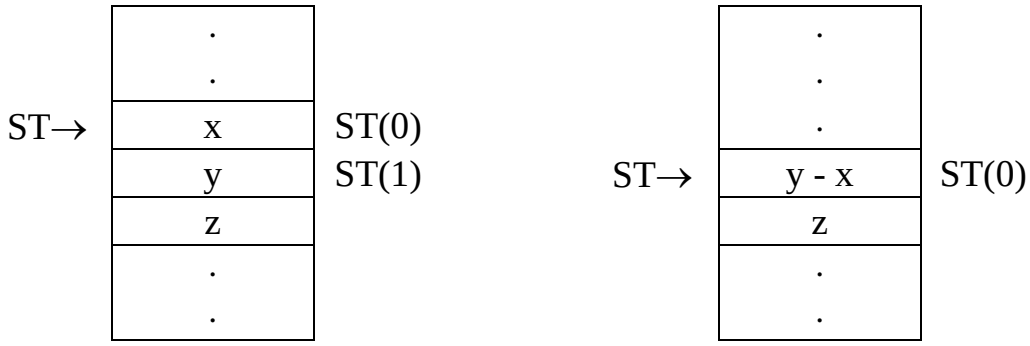


Рисунок 3.12 – Дія команди FSUB

Додаткові арифметичні команди. Ці команди без явних операндів виконують дії над вмістом вершини стека, результат поміщають туди ж без модифікації вказівника стека.

FABS – знаходження абсолютної величини.

FCHS – зміна знака операнда.

FRNDINT – округлення операнда до цілого в форматі з плаваючою точкою.

FSQRT – добування квадратного кореня.

FPREM – обчислення залишок від ділення вмісту ST(0) на число з ST(1). Залишок заміщає число в ST(0).

FSCALE – масштабування на степінь числа 2 – додає ціле число з ST(1) до порядку в регістрі ST(0), тобто множить (або ділить) ST(0) на число $2^{(ST(1)}$. Цю команду можна використовувати для піднесення числа 2 до цілого степеня (додатного або від'ємного).

FXTRACT – розкладає вміст ST(0) на два числа: незміщений порядок (заміняє старе значення в ST (0)) і знакову мантису (включається зверху, тобто в ST(7)).

Команда FSCALE, що знаходиться після команди FXTRACT, відновлює вихідне число.

Команди порівнянь. FCOM ST(i)/mem порівнює вміст ST(0) з операндом в чисельному регістрі або в пам'яті, тобто проводить віднімання операндів без запам'ятовування результату і встановлює коди умов в регістрі стану (табл. 20).

FICOM mem порівнює вмісти моє вершини стека ST(0) з цілим числом у пам'яті.

FCOMPST(i)/mem аналогічна команді FCOM, але після порівняння виробляє вилучення операнда з вершини стека.

FCOMPP ST(i) порівнює ST(0) з ST(i) і витягує з стека обидва операнди.

FTST порівнює вершину стека з нулем.

FXAM порівнює вершину стека з нулем, але виставляє 4 прапора умов (зокрема, визначається ненормалізована мантиса, нескінченність, не число та ін.).

FCOMIST(0), ST(i) порівняння дійсних чисел і установка прапорів в EFLAGS (P6 +).

FCOMIP ST (0), ST (i) порівняння дійсних чисел і установка прапорів в EFLAGS і вилучення операнда з вершини стека (P6 +).

Таблиця 3.4 – Коди умов після порівняння

C3	C0	Умова
0	0	ST(0) >*x*
0	1	ST(0) <*x*
1	0	ST(0) = *x*
1	1	ST(0) і *x* – непорівнянні

Прапори умов (C0, C3) співпроцесора x87 використовуються для організації умовних переходів мікропроцесором x86. Для цього командою FSTSW AX вміст регістра стану x87 копіюється в акумулятор AX мікропроцесора x86. Після цього командою SAHF старший байт акумулятора (AH) передається в молодший байт регістра прапорів. При цьому умові C0 відповідає прапор CF, а умові C3 – прапор ZF.

Трансцендентні команди. Елементарними трансцендентними функціями є:

– тригонометричні функції (sin, cos, tg, ctg);
– обернені тригонометричні функції (arcsin, arccos, arctg, arcctg);
– логарифмічні функції ($\log_2 x$, $\log_{10} x$, $\log_e x$);
– показникові функції (y^x, 2^x, 10^x, e^x);
– гіперболічні функції (sh, ch, th, cth);
– обернені гіперболічні функції (arsh, arch, arth, arcth).

Таблиця 3.5 – Трансцендентні команди

Мнемоніка	Описання команди	Обчислюється функція
FPTAN	Частковий тангенс	ST(1) / ST(0) = tg (ST(0))
FSIN	Синус (387+)	ST(0) = sin (ST(0))
FCOS	Косинус (387+)	ST(0) = cos (ST(0))
FSINCOS	Синус, косинус (387+)	ST(7) = sin (ST(0)); ST(0) = cos (ST(0))
FPATAN	Частковий арктангенс	ST(0) = arctg (ST(1)/ST(0)
FYL2X	Логарифм за основою 2	ST(0) = ST(1) × $\log_2$ (ST(0))
FYL2XP1	Логарифм за основою 2	ST(0) = ST(1) × $\log_2$ (ST(0)+1)
F2XM1	Показникова функція	ST(0) = $2^{(ST(0))} - 1$

Співпроцесор x87 обчислює будь-яку з елементарних трансцендентних функцій від аргументів подвійної точності, даючи результат подвійної точності з помилкою молодшого розряду округлення.

Команда **FPTAN** знаходження часткового тангенса як результат видає два числа (співпроцесори 87/287):

$$y / x = \text{tg}(\text{ST}(0))$$

Число «*y*» замінює старий вміст ST(0), а число «*x*» включається зверху. Тому, після виконання команди вказівник стека зменшиться на 1, число «*x*» буде записане в нову вершину стека ST(0), а число «*y*» – в регістр ST(1).

Для отримання значення тангенса необхідно виконати команду FDIV. Дві команди FPTAN і FDIV вибирають аргумент з вершини стека і туди ж поміщають значення ***тангенса*** (без модифікації вказівника вершини стека). Дві команди FPTAN і FDIVR обчислюють значення ***котангенса***.

Для команди FPTAN ***аргумент задається в радіанах*** і повинен знаходиться в діапазоні (співпроцесори 87/287):

$$0 \le \text{ST}(0) \le 1/4.$$

Для співпроцесорів 387+ аргумент команди FPTAN (***в радіанах***) може бути будь-яким:

$$-2^{63} \le \text{ST}(0) \le +2^{64}.$$

Значення тангенса початкового кута tg(ST(0)) заміщає аргумент і в стек включається зверху 1,0 (для програмної сумісності з попередніми співпроцесорами 87/287).

Значення інших тригонометричних функцій (для співпроцесорів 87/287) можна обчислити, використовуючи формули тангенса половинного кута (табл. 3.6). Тому перед початком обчислення тригонометричних функцій з використанням команди FPTAN необхідно аргумент в ST(0) поділити на 2. Нове значення аргументу «z» має також задовольняти умові:

$$0 \le z \le 1/4 .$$

Таблиця 3.6 – Формули для обчислення тригонометричних функцій

$\sin z = \dfrac{2\cdot(y/x)}{1+(y/x)^2}$	$\cos z = \dfrac{1-(y/x)^2}{1+(y/x)^2}$
$\mathrm{tg}(\mathrm{ST}(0)) = y/x$	$\mathrm{ctg}(ST(0)) = x/y$
$\sec z = \dfrac{1+(y/x)^2}{2\cdot(y/x)}$	$\mathrm{cosec}\, z = \dfrac{1+(y/x)^2}{1-(y/x)^2}$

У співпроцесорах 387+ з'явилися нові команди:

– **FSIN** обчислення синуса;

– **FCOS** обчислення косинуса;

– **FSINCOS** обчислення синуса і косинуса.

Всі вони сприймають в ST(0) вихідний кут, що ***вимірюється в радіанах***, який перебуває в діапазоні: $-2^{63} \le \mathrm{ST}(0) \le +2^{63}$. Команди FSIN і FCOS повертають результат на місце аргументу, а команда FSINCOS повертає значення синуса на місце аргументу і включає значення косинуса в стек.

Команда **FPATAN** обчислює arctg(ST(1) / ST(0)). Два операнда вилучаються з стека, а результат включається в стек. Тому остаточно, вказівник стека збільшується на 1. Операнди цієї команди для співпроцесорів 8087/287 повинні задовольняти умові:

$$0 < \mathrm{ST}(1) < \mathrm{ST}(0) .$$

У співпроцесорах 387+ обмежень на діапазон допустимих аргументів команди FPATAN не існує.

Для обчислення інших обернених тригонометричних функцій за аргументом «z» необхідно попередньо підготувати операнди в ST(0) і ST(1) відповідно до табл. 3.7 (***ділити операнди не потрібно***).

Команда **FYL2X** обчислює функцію: $Y = \mathrm{ST}(1)\log_2\mathrm{ST}(0)$. Два операнди вилучаються з стека, а потім результат включається в стек. Тому вказівник стека збільшиться на 1. У команді потрібне задоволення властивості логарифмічної функції ST(0)>0.

Значення інших логарифмічних функцій обчислюються за формулами (табл. 3.8) із завантаженням в регістр ST(1) необхідних констант командами: FLDLN2 і FLDLG2.

Таблиця 3.7 – Формули для обчислення обернених тригонометричних функцій

$\arcsin z = \operatorname{arctg}\left(\frac{z}{\sqrt{(1-z^2)}}\right)$	←ST(1) ←ST(0)
$\arccos z = 2\cdot\operatorname{arctg}\left(\frac{\sqrt{(1-z)}}{\sqrt{(1+z)}}\right)$	←ST(1) ←ST(0)
$\operatorname{arctg} z = \operatorname{arctg}\left(\frac{z}{1}\right)$ ←ST(1) ←ST(0)	$\operatorname{arcctg} z = \operatorname{arctg}\left(\frac{1}{z}\right)$ ←ST(1) ←ST(0)
$\operatorname{arccosec} z = \operatorname{arctg}\left(\frac{\operatorname{sign} z}{\sqrt{(z^2-1)}}\right)$	←ST(1) ←ST(0)
$\operatorname{arcsec} z = 2\cdot\operatorname{arctg}\left(\frac{\sqrt{(\lvert z\rvert-1)}}{\sqrt{(\lvert z\rvert+1)}}\right)$	←ST(1) ←ST(0)

Таблиця 3.8 – Формули для обчислення логарифмічних функцій

$\log_2 x$ →FLD1; FLD *x*; FYL2X;
$\ln x = \ln 2 \log_2 x$ →FLDLN2; FLD *x*; FYL2X;
$\lg x = \lg 2 \log_2 x$ → FLDLG2; FLD *x*; FYL2X.

Ще одна логарифмічна команда **FYL2XP1** обчислює функцію: $Y=ST(1)\times\log_2(ST(0)+1)$. Причина появи цієї команди полягає в отриманні вищої точності обчислення функції: $\log(1+x)$. Ця функція часто зустрічається в фінансових розрахунках, а також при обчисленні обернених гіперболічних функцій.

Команда показникової функції **F2XM1** обчислює:

$$Y=2^{(ST(0))}-1.$$

Аргумент показникової функції повинен знаходиться в діапазоні:

– для співпроцесорів 87/287: $0 \le ST(0) \le 0{,}5$;

– для співпроцесорів 387+: $-1 \le ST(0) \le 1$.

Обчислення функції $2^x - 1$ замість функції 2^x дозволяє уникнути втрати точності, коли аргумент x близький до 0 (а значення функції 2^x близьке до 1). Інші показникові функції обчислюються за формулами в табл. 3.9.

Константи $\log_2 e$ і $\log_2 10$ вже є в співпроцесорі і завантажуються відповідними командами.

Таблиця 3.9 – Формули для обчислення показникових функцій

$2^x = \left(2^x - 1\right) + 1 = \text{F2XM1}\left(x\right) + 1;$
$e^x = 2^{x\log_2 e} = \text{F2XM1}\left(x\log_2 e\right) + 1;$
$10^x = 2^{x\log_2 10} = \text{F2XM1}\left(x\log_2 10\right) + 1;$
$a^x = 2^{x\log_2 a} = \text{F2XM1}\left(x\log_2 a\right) + 1.$

Таблиця 3.10 – Формули для обчислення гіперболічних функцій

Синус гіперболічний	$\text{sh}\, x = \frac{\text{sign}\, x}{2} \cdot \left[\left(e^{\lvert x\rvert} - 1\right) + \frac{e^{\lvert x\rvert} - 1}{e^{\lvert x\rvert}}\right]$
Косинус гіперболічний	$\text{ch}\, x = \frac{1}{2} \cdot \left(e^{\lvert x\rvert} + \frac{1}{e^{\lvert x\rvert}}\right)$
Тангенс гіперболічний	$\text{th}\, x = \text{sign}\, x - \left[\frac{e^{\left(2\cdot\lvert x\rvert\right)} - 1}{e^{\left(2\cdot\lvert x\rvert\right)} + 1}\right]$
Котангенс гіперболічний	$1/\text{th}\, x$
Косеканс гіперболічний	$1/\text{sh}\, x$
Секанс гіперболічний	$1/\text{ch}\, x$

Таблиця 3.11 – Формули для обчислення обернених гіперболічних функцій

$\text{arsh}\, x = \text{sign}\, x \cdot \ln 2 \cdot \log_2(1+z),\ \text{де}\ z = \lvert x\rvert + \frac{\lvert x\rvert}{\left(1/\lvert x\rvert\right) + \sqrt{1 + \left(1/\lvert x\rvert\right)}}$
$\text{arch}\, x = \ln 2 \cdot \log_2(1+z),\ \text{де}\ z = x - 1 + \sqrt{\left(x^2 - 1\right)}$
$\text{arth}\, x = \text{sign}\, x \cdot \ln 2 \cdot \log_2(z), \text{де}\ z = \frac{2 \cdot \lvert x\rvert}{1 - \lvert x\rvert}$
$\text{arcth}\, x = \text{arth}\left(1/x\right)$
$\text{arcsh}\, x = \text{arsh}\left(1/x\right)$
$\text{arsch}\, x = \text{arch}\left(1/x\right)$

3.3.2 Команди управління співпроцесором x87. Команди управління співпроцесором x87 забезпечують доступ до нечислових регістрів. Мнеміоніки, що починаються з FN, відповідають командам «без очікування», тобто процесор x86 передає їх для виконання в співпроцесор x87, не перевіряючи зайнятість співпроцесора і ігноруючи чисельні особливі випадки.

Мнемоніки без літери «N» відповідають командам «з очікування», тобто змушують процесор x86 реагувати на незамасковані особливі випадки і чекати завершення виконання команд в співпроцесорі x87. У загальному випадку, програмістам рекомендується уникати форм команд «без очікування».

Команда FNSTCW mem (FSTCW mem) передає вміст регістра управління (CW) в комірку пам'яті.

Команда FLDCW mem завантажує регістр управління (CW) з комірки пам'яті.

Ці дві команди застосовуються для зміни режиму роботи співпроцесора x87.

Команда FNSTSW mem (FSTSW mem) передає вміст регістра стану (SW) співпроцесора x87 в комірку пам'яті.

Команда FNSTSW AX (FSTSW AX) передає вміст регістра стану (SW) співпроцесора в регістр AX мікропроцесора x86.

Команда FNCLEX (FCLEX) скидає в регістрі стану співпроцесора прапори особливих випадків, а також біти ES і BUSY. Ці прапори не скидаються апаратно і повинні явно скидатися програмістом.

Команда FNINIT (FINIT) ініціалізує регістри управління, стану і тегів на значення, наведені в табл. 3.12. Таку ж дію здійснює апаратний сигнал скидання RESET.

Таблиця 3.12 – Ініціалізація співпроцесора x87

Регістр	Вибір	Режим роботи
Регістр управління	Режим нескінченності	Проективний – (287) Афінний – (387+)
	Режим округлення	Округлення до найближчого
	Точність	Розширена
	Усі особливі випадки	Замасковані
Регістр стану	Біт зайнятості	B = 0: не зайнятий
	Код умови	Не визначений
	Вказівник стека	TOP = 000
	Біт сумарної помилки	ES = 0
Регістр тегів		Усі теги показують – «порожній»

Лабораторна робота № 6 Обчислення значення функції з використанням FPU

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: обчислити значення функції:

$$S = \frac{1}{2} * A * B * \sin(\alpha)$$

```
void main ()                  // початок програми мовою C++
{
long  A=20, B=30, M2=2;       // опис операндів у пам'яті
float       ALPHA=0.7, Y;

__asm{                  ; початок асемблерної вставки
finit                         ; очищення регістрів співпроцесора
     fld     ALPHA            ; завантаження у вершину стека аргументу
     fsin                     ; обчислення синуса
     fimul   A                ; множення вершини стека на константу
     fimul   B
     fidiv   M2
     fstp    Y                ; збереження результату в комірці пам'яті
}                       // закінчення асемблерної вставки
}                             // закінчення програми мовою C++
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою)
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Обчислити значення функції:

Варіант 1. $Y = 3,5 * x^2 + 7,2 * x + 24$

Варіант 2. $a_n = 2,5 * n^2 + 5,3*n - 21$

Варіант 3. $Y = 2500 / (2 * x^2 + 3,7)$

Варіант 4. $Y = \sqrt{14 * x^2 + 5,6}$

Варіант 5. $Y = (20 * x) /(5 * x^2 – 8,5)$

Варіант 6. $a_n = (4,5^n) / (n + 5,1)$

Варіант 7. $Y = (x - 7) / \sqrt{x^2 + 20}$

Варіант 8. $Y = 1024 / (3,2 * x^2 – 25)$

Варіант 9. $a_n = 418 / (2 * n^2 + 7,3$

Варіант 10. $Y = (3000+x) / (x^2 + 2,6 * x – 7,5)$

Варіант 11. $Y = 3,5 * x^2 + 7,2 * x - 2,2$

Варіант 12. $a_n = 2,5 * n^2 + 5,3* n +5$

Варіант 13. $Y = 2500.12 / (2 * x^2 + 3,7*x – 8,1$

Варіант 14. $Y = \sqrt{14 * x^2 + 5,6 * x - 20}$.

Варіант 15. $Y = (20 * x - 15) /(5 * x^2 – 8,5*x + 15,2$

Варіант 16. $a_n = (4,5^n - 1) / (n + 5.4)$

Варіант 17. $Y = (x - 7) / \sqrt{x^2 + 20 * x - 24}$

Варіант 18. $(1024 – 2 * x^2)/ (3,2 * x^2 – 25)$

Варіант 19. $a_n = (418 + n) / (2 * n^2 + 2,3)$

Варіант 20. $Y = (3000 – 2x^2 + 2,5x)/ (x^2 + 3,6 * x – 7,5)$

Варіант 21. $Y = (3,5 * x^2 + 7,2 * x +24)/(x^2-23,4)$

Варіант 22. $a_n = (2,1 * n^2 - 2,3*n – 21) /(n+1,4)$

Варіант 23. $Y = 4000 / (2 * x^2 + 3,7*x - 4,3)$

Варіант 24. $Y = \sqrt{14 * x^2 + 5,6} * x - 2,4$

Варіант 25. $Y = (1124,4 – x) / (3,2 * x^2 – 25)$

Лабораторна робота № 7 Умовні переходи FPU

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: визначити номер (n) елемента прогресії: $a_n = 5{,}3^n + 5 * n$, при якому сума елементів перевищуватиме 20000.

```
void main ()                        // початок програми на  C++
{
float       A=5.3;           // опис операндів у пам'яті
long  B=5, C=20000, N=0;

__asm{                 ; початок асемблерної вставки
finit                        ; очищення регістрів сопроцесора
fld1                   ; регістр для обрахування степеневої функції.
     fldz                    ; регістр для накопичення суми прогресії
m1:  inc          N           ;нарощування аргументу
fld        A           ; завантаження в ST(0) 5,3
     fmulp        ST(2),ST    ; обчислення степеневої функції  5,3^n
     fild         B
     fimul        N
     fadd         ST,ST(2)    ; очищення елементу прогресії
     fadd                     ; накопичення суми прогресії
     ficom        C           ; порівняння суми з 20000
                   fstsw      AX          ; збереження регістру стану  SW
           (FPU) в
                          ; регістрі  AX (CPU)
     sahf                     ;збереження старшого байту AX в рег. флагів
     jc           m1          ; перехід, якщо сума менша 20000
     }                        // закінчення асемблерної вставки
}                             // закінчення програми на  C++
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі або в іншій програмі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою)
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант 1. Знайти ціле значення аргументу, при якому функція $Y = 20 / (x^2 + 2{,}5^x)$ стане менше 0,2.

Варіант 2. Знайти ціле значення аргументу x, при якому функція $Y = 15 / (x^2 + 3{,}7)$ стане менше 0,1 (для x – від 1 із кроком 1).

Варіант 3. Визначити номер (n) елемента прогресії $a_n = 2{,}5 * n^2 + 7{,}3$, при якому сума елементів прогресії перевищить 1000.

Варіант 4. Визначити номер (n) елемента прогресії $a_n = 3{,}3^n + 5$, при якому сума елементів перевищить 15000.

Варіант 5. Заданий масив з елементами $a(i) = \sin (5 * i)$. Визначити номер елемента масиву, при якоум сума елементів перевищить 3. Аргумент синуса заданий у градусах.

Варіант 6. Знайти ціле значення аргументу, при якому функція $Y = \sqrt{2 * 3{,}5^x + 10}$ перевищить 100.

Варіант 7. Визначити номер (n) елемента прогресії $a_n = \sqrt{2{,}5^n + 3 * n}$, при якому сума елементів перевищить 100.

Варіант 8. Заданий масив з елементами $b(i) = \sin(2 * i^2)$. Визначити номер елемента, при якому сума елементів перевищить 3. Аргумент синуса заданий у градусах.

Варіант 9. Знайти ціле значення аргументу, при якому функція $Y = (5{,}6^x) / (3 * x^2)$ перевищить 200.

Варіант 10. Знайти ціле значення аргументу x, при якому функція $Y = \sqrt{15 * x^2 + 32 * x + 40}$ перевищить 30.

Варіант 11. Знайти ціле значення аргументу, при якому функція $Y = 30/ (1+x^2 + 2{,}5^x)$ стане менше 0,2.

Варіант 12. Знайти ціле значення аргументу x, при якому функція $Y = 25 / (x^2 - 3{,}7)$ стане менше 0,1 (для x – від 1 із кроком 1).

Варіант 13. Визначити номер (n) елемента прогресії $a_n = 2{,}5 * n^2 + 7{,}3 * n - 1$, при якому сума елементів прогресії перевищить 2000.

Варіант 14. Визначити номер (n) елемента прогресії $a_n = 3{,}3^n + 5 * n - 5$, при якому сума елементів перевищить 15000.

Варіант 15. Заданий масив з елементами $a(i) = 2*\cos (5 * i)$. Визначити номер елемента масиву, при якому сума елементів перевищить 3. Аргумент косинуса заданий у градусах.

Варіант 16. Знайти ціле значення аргументу, при якому функція $Y = \sqrt{4 * 3{,}5^x + 10 * x - 2}$ перевищить 120.

Варіант 17. Визначити номер (n) елемента прогресії $a_n = \sqrt{2{,}5^n + 3{,}3 * n - 4}$, при якому сума елементів перевищить 100.

Варіант 18. Заданий масив з елементами $b(i) = \cos(2 * i^2)$. Визначити номер елемента, при якому сума елементів перевищить 3. Аргумент косинуса заданий у градусах.

Варіант 19. Знайти ціле значення аргументу, при якому функція $Y = (2,3+5,6^x) / (2+3 * x^2)$ перевищить 400.

Варіант 20. Знайти ціле значення аргументу x, при якому функція $Y = \sqrt{1,5 * x^2 + 31,6 * x + 45}$ перевищить 20.

Варіант 21. Знайти ціле значення аргументу, при якому функція $Y = (40*x-4,3) / (x^2 + 2,5^x)$ стане менше 0,1.

Варіант 22. Знайти ціле значення аргументу x, при якому функція $Y = (15*x -14) / (x^2 + 3,7)$ стане менше 0,5 (для x – від 1 із кроком 1).

Варіант 23. Визначити номер (n) елемента прогресії $a_n = 2,5 * n^2 + 7,3 * n - 25$, при якому сума елементів прогресії перевищить 800.

Варіант 24. Визначити номер (n) елемента прогресії $a_n = 3,3^n + 5*n$, при якому сума елементів перевищить 35000.

Варіант 25. Заданий масив з елементами $a(i) =2* sin (5 * i -1,32)$. Визначити номер елемента масиву, при якому сума елементів перевищить 5. Аргумент синуса заданий у градусах.

Лабораторна робота № 8 Тригонометричні функції FPU

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: обчислити функцію Y. Результат перевести в градуси

$$Y = 5 * arcsin ((tg\ 60°)^{2/7}).$$

```
void main ()                        // початок програми мовою З++
{
long  A=60, B=5, C=7, D=180; // опис операндів у пам'яті
float       Y;

__asm{                  ; початок асемблерної вставки
finit                         ; очищення регістрів співпроцесора
fldpi                         ; завантаження у вершину стека числа 3,1415…
fimul           A             ; множення числа «пі» на аргумент
fidiv           D             ; розподіл аргументу на 180
fptan                         ; обчислення часткового тангенса
fdiv                    ; знаходження тангенса
fmul       ST,ST ; зведення у квадрат
fidiv           C             ; обчислений «z» - аргумент Arcsin
fld        ST           ; копіювання вершини стека
fmul       ST,ST ; зведення у квадрат
fld1                    ; включення в стек «1»
fsubr                         ; вирахування з реверсом : 1 – z²
fsqrt                         ; корінь квадратний
fpatan                        ; обчислення арктангенса
fimul           B             ; множення на константу
fldpi                         ;
fdiv                    ;     переведення з радіанів – у градуси
fimul           D             ;
fstp       Y            ; збереження результату в комірці пам'яті
}                       // закінчення асемблерної вставки
}                             // закінчення програми мовою C++
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі або в іншій програмі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою)
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант 1. $Y = 3 * \arcsin (2 * \cos(70°))^2$.
Варіант 2. $Y = (1/2) * \text{arccosec} (3 * A^2 + 4 * B)$.
Варіант 3. $Y = 2 * \arcsin ((A^2 + B^2)/3)$.
Варіант 4. $Y = 3 * \arccos (2 * A^2 - B)$.
Варіант 5. $Y = 5 * \text{arcsec} (3 * (\text{tg } 70°)^2)$.
Варіант 6. $Y = (1/3) \arcsin (3 * A + \sin 20°)$.
Варіант 7. $Y = 4 * \arccos (2 * \sin 30° * \cos 30°)$.
Варіант 8. $Y = (1/4) \text{ arccosec } ((5 * A + B^2)/2)$.
Варіант 9. $Y = 3 * \text{arccosec} (4 * A + \text{tg } 40°)$
Варіант 10. $Y = 5 * \arcsin (3 * \text{tg } 25° * \sin 25°)$.
Варіант 11. $Y = 3 * \arcsin (2 * \sin(70°))^2$.
Варіант 12. $Y = (1/4) * \text{arcsec} (3 * A^2 + 4 * B)$.
Варіант 13. $Y = 4 * \arccos ((A^2 + B^2)/3)$.
Варіант 14. $Y = 3 * \arcsin (2 * A^2 - B)$.
Варіант 15. $Y = 5 * \text{arcsec} (3 * (\sin 70°)^2)$.
Варіант 16. $Y = (1/3) \arcsin (3 * A + \sin 20°)$.
Варіант 17. $Y = 2 * \text{arctg} (2 * \sin 30° * \cos 30°)$.
Варіант 18. $Y = (1/4) \text{ arccosec } ((5 * A + B^2)/2)$.
Варіант 19. $Y = 1,5 * \text{arcsec} (4 * A + \text{tg } 30°)$
Варіант 20. $Y = 2 * \arccos (3 * \cos 25° * \sin 25°)$.
Варіант 21. $Y = 3,4 * \arcsin (2 * \cos(40°))^2$.
Варіант 22. $Y = (1/3) * \text{arccosec} (1,2 * A^2 + 4 * B)$.
Варіант 23. $Y = 2* A * \arccos ((A^2 + B^2)/3)$.
Варіант 24. $Y = 3,8 * \arcsin (2 * A^2 – B - C)$.
Варіант 25. $Y = 5,1 * \text{arcsec} (3 * (\sin 70°)^2)$.

Лабораторна робота № 9 Логарифмічні та показникові функції FPU

1. У Microsoft Visual Studio створити проект консольного додатку на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: обчислити функцію Y. Результат перевести в градуси.

$$Y = 5 * arcsin ((tg\ 60°)^{2/7}).$$

```
void main ()                        // початок програми мовою З++
{
long  A=60, B=5, C=7, D=180; // опис операндів у пам'яті
float       Y;

__asm{                  ; початок асемблерної вставки
finit                         ; очищення регістрів співпроцесора
fldpi                         ; завантаження у вершину стека числа 3,1415…
fimul           A             ; множення числа «пі» на аргумент
fidiv           D             ; розподіл аргументу на 180
fptan                         ; обчислення часткового тангенса
fdiv                    ; знаходження тангенса
fmul       ST,ST ; зведення у квадрат
fidiv           C             ; обчислений «z» – аргумент Arcsin
fld        ST           ; копіювання вершини стека
fmul       ST,ST ; зведення у квадрат
fld1                    ; включення в стек «1»
fsubr                         ; вирахування з реверсом : 1 – z²
fsqrt                         ; корінь квадратний
fpatan                        ; обчислення арктангенса
fimul           B             ; множення на константу
fldpi                         ;
fdiv                    ;     переведення з радіанів – у градуси
fimul           D             ;
fstp       Y            ; збереження результату в комірці пам'яті
}                       // закінчення асемблерної вставки
}                             // закінчення програми мовою C++
```

4. Для покрокової відладки програми необхідно: встановити курсор на початку першого рядка асемблерної вставки та натиснути «CTR-F10». Кожен крок здійснюється натисканням клавіші «F10».
5. В прикладі реалізувати виведення результату на екран.
6. Порівняти результат виконання програми з обрахунком значення функції на калькуляторі або в іншій програмі.
7. Виконати індивідуальне завдання.
8. Звіт з лабораторної роботи повинен містити:
   a. Титульний лист.
   b. Виконаний приклад.
   c. Тест виконання прикладу (порівняння результату з іншою програмою)
   d. Вихідний код індивідуального завдання.
   e. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

Варіант 1. Обчислити 6 значень функції $Y = 5 * ln (sin x)$, x змінюється в градусах від 10 із кроком 15.

Варіант 2. Обчислити 5 значень функції $Y = 7^x$, x змінюється від 0,5 із кроком 0,2.

Варіант 3. Обчислити 7 значень функції $Y = 4 * lg (tg x)$, x змінюється в градусах від 15 із кроком 10.

Варіант 4. Обчислити 6 значень функції $Y = 12^x$, x змінюється від 0,5 із кроком 0,3.

Варіант 5. Обчислити 5 значень функції $Y = 3 * log_8(x^2 + 1)$, x змінюється від 0,2 із кроком 0,3.

Варіант 6. Обчислити 7 значень функції $Y = 5^{(sin\ x)}$, x змінюється в градусах від 10 із кроком 8.

Варіант 7. Обчислити 6 значень функції $Y = 7 * \ln(x^2 + \sqrt{x})$, x змінюється від 2 із кроком 3.

Варіант 8. Обчислити 5 значень функції $Y = 4(x^2 + 1)$, x змінюється від 0,2 із кроком 0,4.

Варіант 9. Обчислити 7 значень функції $Y = 6 * lg (cos x)$, x змінюється в градусах від 8 із кроком 12.

Варіант 10. Обчислити 6 значень функції $Y = 3^{(cos\ x)}$, x змінюється в градусах від 10 із кроком 8.

Варіант 11. Обчислити 8 значень функції $Y = 3 * ln (4*sin\ 2*x)$, x змінюється в градусах від 10 із кроком 15.

Варіант 12. Обчислити 10 значень функції $Y = 7^{x+3}$, x змінюється від 0,5 із кроком 0,2.

Варіант 13. Обчислити 4 значень функції $Y = 4 * lg (ctg\ 2*x)$, x змінюється в градусах від 15 із кроком 10.

Варіант 14. Обчислити 9 значень функції $Y = 4 + 12^{x-2}$, x змінюється від 0,5 із кроком 0,3.

Варіант 15. Обчислити 5 значень функції $Y = 3 * log_8(x^2 + 1)$, x змінюється від 0,2 із кроком 0,3.

Варіант 16. Обчислити 12 значень функції $Y = 1+5^{(sh\ x)}$, x змінюється в градусах від 10 із кроком 8.

Варіант 17. Обчислити 6 значень функції $Y = 7{,}3 * \ln(2 * x^2 + \sqrt{x})$, x змінюється від 2 із кроком 3.

Варіант 18. Обчислити 5 значень функції $Y = ln(4(x^2 + 1)/(x-2))$, x змінюється від 0,2 із кроком 0,4.

Варіант 19. Обчислити 8 значень функції $Y = 6 * ln (sin x)$, x змінюється в градусах від 5 із кроком 8.

Варіант 20. Обчислити 6 значень функції $Y = 3^{(cos(2-5,1*x)}$, x змінюється в градусах від 12 із кроком 8.

Варіант 21. Обчислити 6 значень функції $Y = 4 * lg (cos\ 2x)$, x змінюється в градусах від 4 із кроком 13.

Варіант 22. Обчислити 5 значень функції $Y = ln(25,4*x + 7^x)$, x змінюється від 0,5 із кроком 0,2.

Варіант 24. Обчислити 7 значень функції $Y = 2 * ln\ (ctg\ x)-10,5$, x змінюється в градусах від 12 із кроком 10.

Варіант 24. Обчислити 20 значень функції $Y = 2*cos(1+12^x)$, x змінюється від 0,5 із кроком 0,3.

Варіант 25. Обчислити 7 значень функції $Y = 3 * log_4(7*x^2 + 11)$, x змінюється від 0,5 із кроком 3

Лабораторна робота № 10-11 Ланцюгові операції

Мета роботи. Задавши одномірний масив цілих або дійсних даних A в одному із заданих форматів (short int – INTEGER, long int – LONGINT, float – SINGLE, double), реалізувати обробку масиву, як зазначено у варіанті. Довжина масиву N. Вихідні дані задати самостійно, враховуючи формат елементів масиву A.

У програмі на C++ повинні бути передбачені функції введення-виводу елементів масиву A и його обробки. Вихідні дані повинні вводитися коректно й з перевіркою на область допустимих значень. Тип результату повинен бути дійсним, а його формат визначається з контексту завдання.

Порядок роботи.

1. Уважно вивчити свій варіант обробки елементів масиву.
2. Написати на базовій алгоритмічній мові програму введення вихідних даних (з контролем припустимого діапазону), обробки елементів масиву й виводу отриманого результату.
3. Написати модуль обробки елементів масиву мовою Асемблера.
4. Вмонтувати виклик цього модуля в програму на базовій алгоритмічній мові.
5. Зробити тестові перевірки, відзначити нормальні й аномальні результати, зробити аналіз результатів.

Приклад оформлення звіту наведено у додатку 1.

Індивідуальні завдання

1. Знайти добуток квадратів усіх від'ємних елементів масиву A={a[i]}.
2. Знайти суму перших K від'ємних елементів масиву A={a[i]}.
3. Знайти, добуток усіх від'ємних елементів масиву A={a[i]}, що задовольняють умові: b <= a[i] <= d.
4. Знайти суму кубів усіх від'ємних елементів масиву A={a[i]}.
5. Знайти, суму всіх додатних елементів масиву A={a[i]}, що задовольняють умові: b<= a[i] <= d.
6. Знайти суму квадратів усіх додатних елементів масиву A={a[i]}.
7. Знайти добуток квадратів додатних елементів масиву A={a[i]}, що задовольняють умові: a[i] >= c.
8. Знайти добуток усіх елементів масиву A={a[i]}, що збігаються з його останнім елементом.
9. Знайти добуток квадратів від'ємних елементів масиву A={a[i]}, що задовольняють умові: a[i] <= c.
10. Знайти добуток квадратів останніх L від'ємних елементів у масиві A={a[i]}.
11. Знайти суму перших До елементів масиву A={a[i]}, що задовольняють умові: b<=a[i] <= d.
12. Знайти суму всіх елементів масиву A={a[i]}, що збігаються з його

першим елементом.

13. Знайти суму додатних елементів масиву A={a[i]}, що задовольняють умові: b <= a[i] <= d.

14. Знайти суму останніх L додатних елементів у масиві A={a[i]}.

15. Знайти добуток усіх додатних елементів масиву A={a[i]}, що задовольняють умові: b <= a[i] <= d.

16. Знайти добуток квадратів усіх додатних елементів масиву A={a[i]}.

17. Знайти суму перших K додатних елементів масиву A={a[i]}.

18. Знайти суму кубів усіх додатних елементів масиву A={a[i]}.

19. Знайти добуток усіх від'ємних елементів масиву A={a[i]}, що збігаються з його останнім елементом за модулем.

20. Знайти добуток усіх непарних елементів масиву A={a[i]}

21. Знайти добуток квадратів усіх від'ємних елементів масиву A={a[i]}.

22. Знайти суму перших K від'ємних елементів масиву A={a[i]}.

23. Знайти, добуток усіх від'ємних елементів масиву A={a[i]}, що задовольняють умові: b <= a[i] <= d.

24. Знайти суму кубів усіх від'ємних елементів масиву A={a[i]}.

25. Знайти, суму всіх додатних елементів масиву A={a[i]}, що задовольняють умові: b<= a[i] <= d.

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ПОЛТАВСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ КОНДРАТЮКА

Інститут інформаційних технологій і механотроніки
Кафедра комп’ютерних та інформаційних технологій і систем

Лабораторна робота № 4

з навчальної дисципліни
"КОМП'ЮТЕРНА СХЕМОТЕХНІКА ТА АРХІТЕКТУРА КОМП’ЮТЕРІВ"

Варіант – 6

Виконала:
студентка 201-ТН
Івінська Катерина Дмитрівна
Перевірив:
Демиденко Максим Ігорович

Полтава 2017

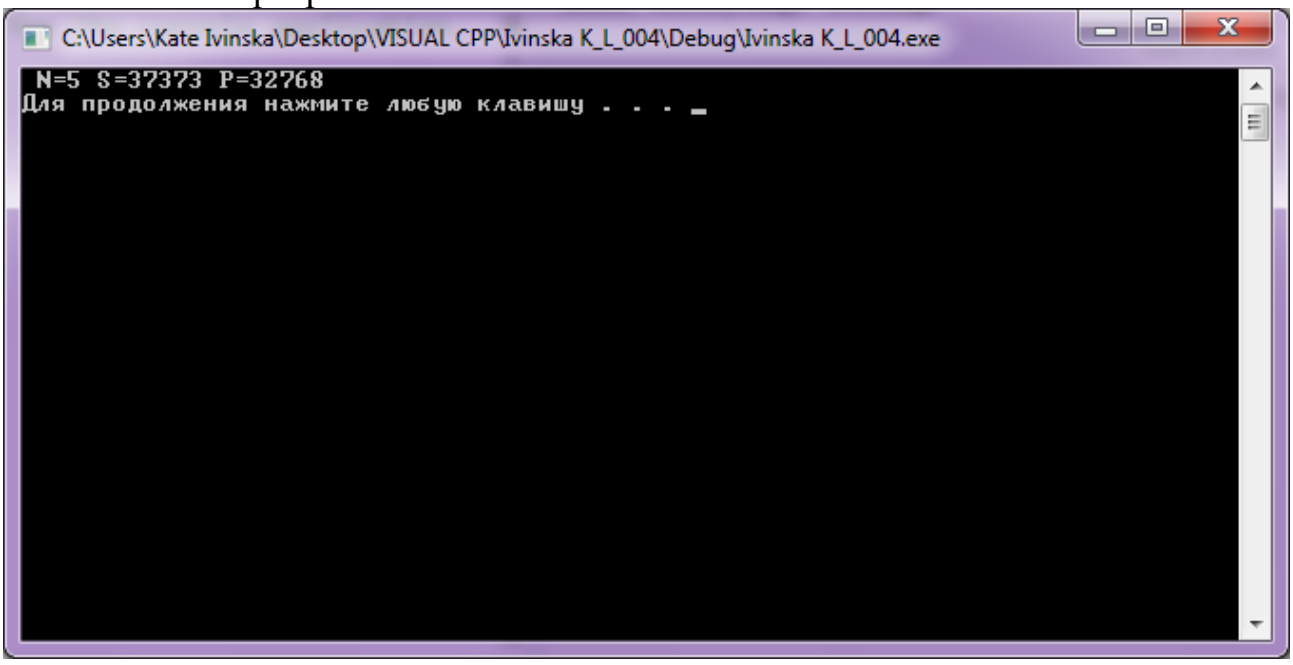
Основна частина

Визначити номер (n) елемента прогресії $a_n = 8n - 5 * n$, при якому сума елементів прогресії перевищить 10000.

Код програми:

```
#include"stdafx.h"
#include<iostream>
usingnamespace std;
void main() // початок програми мовою с++
{
	long N=0;			// змінна пам’яті для аргументу
	long S=0;			//змінна для зберігання суми
	long P=1;			// змінна для нагромадження 8n

	_asm{				; початок асемблерної вставки
		m1: inc N		; збільшення аргументу
		mov EAX, 8		; EAX = 8
		mul P			; множення – 8 n
		mov P, EAX		; пересилання 8n у комірку пам’яті
		add S, EAX		; нагромадження суми
		mov EAX, 5		; EAX = 5
		mul N			; EAX = 5 * n
		sub S, EAX		; нагромадження суми
		cmp S, 10000		; порівняння суми з 10000
		jc m1			; перехід, якщо сума менше 10000
	}				// закінчення асемблерної вставки
}
```

Лістинг програми:

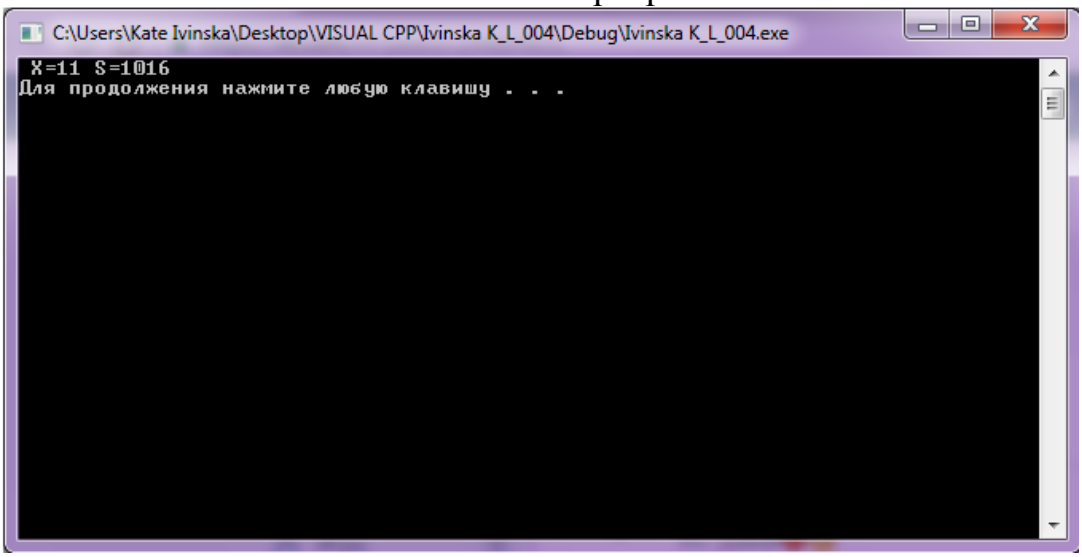


Індивідуальне завдання

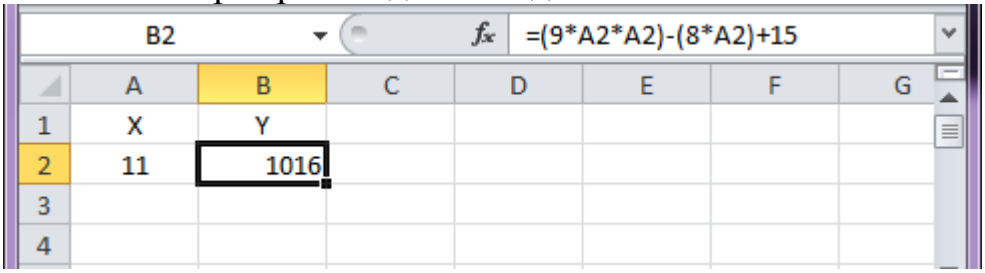
Знайти ціле значення аргументу, при якому функція Y = 9 * x2 – 8 * x + 15 стане більше 1000.

```
#include"stdafx.h"
#include<iostream>
usingnamespace std;
void main() // початок програми мовою с++
{
	long X = 0;			// змінна пам’яті для аргументу
	long S = 0;			//змінна для зберігання суми

	_asm {				; початок асемблерної вставки

		m1 : inc X		; збільшення аргументу
		mov EAX, 9		; EAX = 9
		mul X			; EAX = 9*x
		mul X			; EAX = 9*x*x
		mov S, EAX		; пересилання 9*x*x у комірку пам’яті
		mov EAX, 12		; EAX = 12
		mul X			; EAX = 9 * x
		sub EAX, S		; EAX = 9*x*x - 9*x
		add S, 15		; додавання до S 15
		cmp S, 1000		; порівняння суми з 1000
		jc m1			; перехід, якщо сума менше 10000
		}			// закінчення асемблерної вставки

	cout <<" X="<< X <<" S="<< S << endl;
	system("pause");
}
```

Лістинг програми:



Перевірка завдання за допомогою Excel:



ВИСНОВОК

Для вирішення поставленого завдання було використано інтегроване середовище розробки та тестування програм MS Visual Studio, мову програмування C++ та вставку Assembler та програму Excel з допомогою якої було перевірено правильність вирішення завдання.

ЛІТЕРАТУРА

1. Логічні елементи [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D1%96_%D0%B5%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B8.
2. Тригер [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A2%D1%80%D0%B8%D0%B3%D0%B5%D1%80
3. Регістр (цифрова техніка) [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%96%D1%81%D1%82%D1%80_(%D1%86%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0_%D1%82%D0%B5%D1%85%D0%BD%D1%96%D0%BA%D0%B0)
4. Лічильник імпульсів [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9B%D1%96%D1%87%D0%B8%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D1%96%D0%BC%D0%BF%D1%83%D0%BB%D1%8C%D1%81%D1%96%D0%B2
5. Шифратор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80%D0%B0%D1%82%D0%BE%D1%80
6. Дешифратор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%88%D0%B8%D1%84%D1%80%D0%B0%D1%82%D0%BE%D1%80
7. Мультиплексор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%BE%D1%80
8. Демультиплексор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D0%BC%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%BE%D1%80
9. Суматор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A1%D1%83%D0%BC%D0%B0%D1%82%D0%BE%D1%80
10. IA-32 Intel® Architecture Software Developer's Manual. Vol. 1. Basic architecture. Intel Corporation, 2002.
11. IA-32 Intel® Architecture Software Developer's Manual. Vol. 3. System programming guide. Intel Corporation, 2002.
12. Assembler. Учебник для вузов. 2-е изд. / В. И. Юров – СПб.: Питер, 2003. – 637 с.: ил.
13. Assembler. Практикум. 2-е изд / В. И. Юров – СПб.: Питер, 2003. – 395 с.: ил.
14. Программирование на ассемблере на платформе x86-64 /Р.З. Аблязов – М:ДМК Пресс, 2011. 304 с.: ил.