Міністерство освіти і науки України

Національний університет «Полтавська політехніка
імені Юрія Кондратюка»

Кафедра комп’ютерних та інформаційних технологій і систем

# НАВЧАЛЬНИЙ ПОСІБНИК З ДИСЦИПЛІНИ **«КОМП’ЮТЕРНА СХЕМОТЕХНІКА ТА АРХІТЕКТУРА КОМП’ЮТЕРІВ»** ДЛЯ СТУДЕНТІВ 1 КУРСУ СПЕЦІАЛЬНОСТІ 122 «КОМП’ЮТЕРНІ НАУКИ»

Полтава 2023

Навчальний посібник з дисципліни «Комп’ютерна схемотехніка та архітектура комп’ютерів» для студентів спеціальності 122 «Комп’ютерні науки». – Полтава: НУПП, 2023. – 203 с.

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

Відповідальний за випуск: О.А. Двірна, в.о. завідувачки кафедри комп’ютерних інформаційних технологій і систем, кандидат технічних наук, доцент

Рецензенти: д.т.н., професор Ю.Л. Поночовний
д.т.н., професор О.Л. Ляхов

Під редакцією Руденка О.А.

Затверджено науково-методичною
радою університету
від 14 червня 2023_ р.,
протокол № 5

**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 «Вища математика», «Дискретні структури».

*Програмні компетентності:*

**ЗК1.** Здатність до абстрактного мислення, аналізу та синтезу.

**ЗК2.** Здатність застосовувати знання у практичних ситуаціях.

**ЗК3.** Знання та розуміння предметної області та розуміння професійної діяльності.

**СК12.** Здатність забезпечити організацію обчислювальних процесів в інформаційних системах різного призначення з урахуванням архітектури, конфігурування, показників результативності функціонування операційних систем і системного програмного забезпечення.

*Програмні результати навчання:*

**ПР13.** Володіти мовами системного програмування та методами розробки програм, що взаємодіють з компонентами комп'ютерних систем, знати мережні технології, архітектури комп'ютерних мереж, мати практичні навички технології адміністрування комп'ютерних мереж та їх програмного забезпечення.

*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;

– тестувати й налагоджувати апаратно-програмні засоби і комплекси систем автоматизації та управління.

# 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 $y = f\left(x_1, x_2, \ldots, x_n\right)$ називається перемикальною, або логічною, якщо сама функція $y$ і кожен з її аргументів $x_i$, набувають значень з множини $\{0;1\}$.

Всього існує $x^{x^n \cdot m}$ 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 може бути 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 зібрати не можна.

**Заперечення. Операція «НІ»**

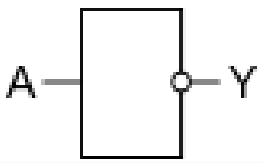



Рисунок 1.1 – Інвертор (IEC)

Рисунок 1.2 – Інвертор (ANSI)

| | |
|---|---|
| 0 | 1 |
| 1 | 0 |

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

**Повторення**

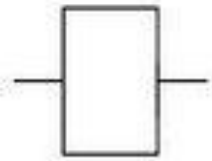


Рисунок 1.3 – Повторювач

| | |
|---|---|
| 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»**

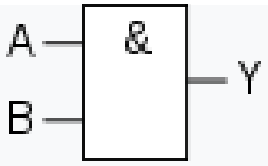



Рисунок 1.4 – Логічний елемент «І» (IEC)

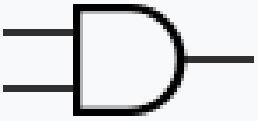


Рисунок 1.5 – Логічний елемент «І» (ANSI)

| | | |
|---|---|---|
| 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. 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»**

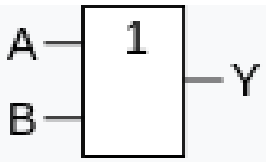



Рисунок 1.6 – Логічний елемент «АБО» (IEC)

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. 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-НІ» (штрих Шефера)**

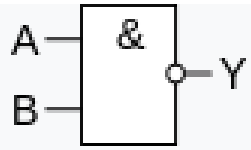



Рисунок 1.8 – Логічний елемент «І-НІ» (IEC)

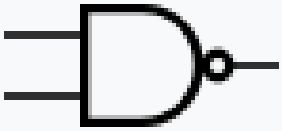


Рисунок 1.9 – Логічний елемент «І-НІ» (ANSI)

| | | |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Мнемонічне правило для І-НІ з будь-якою кількістю входів. 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-НІ» (стрілка Пірса)**

В англомовній літературі NOR.

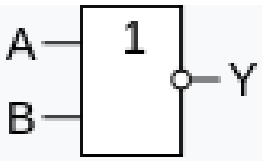



Рисунок 1.10 – Логічний елемент «АБО-НІ» (IEC)

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

| | | ↓ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Мнемонічне правило для АБО-НІ з будь-якою кількістю входів. 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».

**Еквівален2ція. Операція «ВИКЛЮЧЕНЕ_АБО-НІ»**

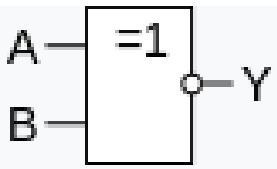



Рисунок 1.12 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО-НІ» (IEC)

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

| | | ↔ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Мнемонічне правило ВИКЛЮЧНЕ_АБО-НІ з будь-якою кількістю входів. 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**

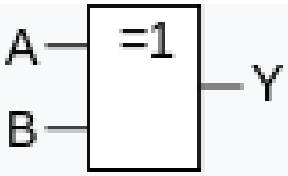



Рисунок 1.14 – Логічний елемент «ВИКЛЮЧЕНЕ_АБО» (IEC)

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. 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 на виході, що містять елементарну запам’ятовувальну комірку – бістабільну схему (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 тощо, 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

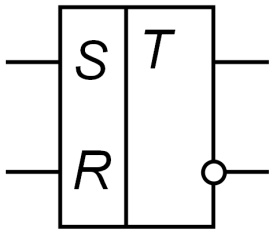



Рисунок 1.17 – Умовне графічне позначення асинхронного RS-тригера

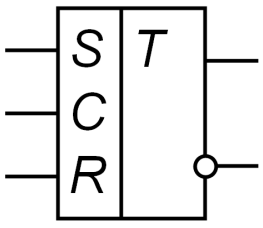



Рисунок 1.18 – Умовне графічне позначення синхронного RS-тригера

| $C$ | $S$ | $R$ | $Q(t)$ | $Q(t+1)$ |
|---|---|---|---|---|
| 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

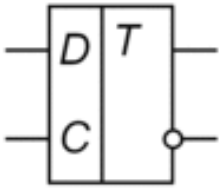



Рисунок 1.20 – Умовне графічне позначення синхронного D-тригера із статичним входом синхронізації C

**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. Двоступінчастий 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

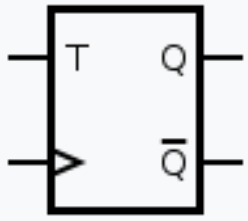



Рисунок 1.22 – Умовне графічне позначення синхронного Т-тригера з динамічним входом синхронізації С на схемах

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 двоступінчастому (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 двоступінчастому (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 **Т** вхід подають одиницю, а на **С**– 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**

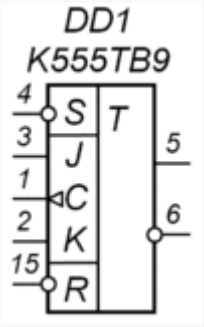



Рисунок 1.24 – Умовне графічне позначення JK-тригера з додатковими асинхронними інверсними входами **S** і **R**

**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 T-тригер, об’єднавши входи **J** і **K** [2].

Алгоритм функціонування JK-тригера можна представити формулою:

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

## 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].

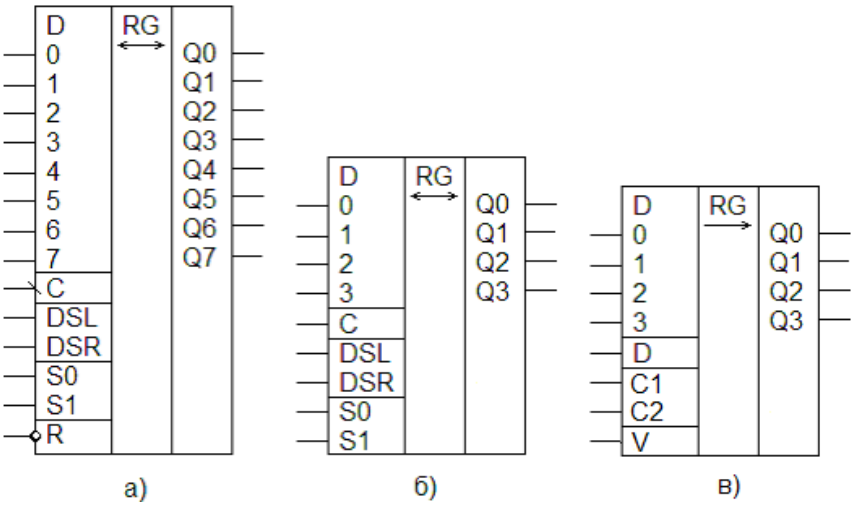



Рисунок 1.25 – Універсальні регістри зсуву: а – К155ІР13, б – К500ІР141, в – КМ155ІР1

## 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 $M$ знаходять із виразу $n = \log_2 M$. Значення поточного числа вхідних імпульсів $n$-розрядного підсумовуючого лічильника при відліку з нульового початкового стану визначають за формулою:

$$N^+ = \sum_{i=1}^{n} 2^{i-1} Q_i = 2^{n-1} Q_n + 2^{n-2} Q_{n-1} + \ldots + 2^0 Q_1,$$

де $2^{i-1}$ – вага $i$-го розряду; $Q_i \in \{0;1\}$ – 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 $P_i = U^+ Q_i Q_{i-1} \ldots Q_1 = 1$. 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

$$\sum U^+ - \sum U^- = N - N_П,$$

де $N$ – значення коду на прямих виходах тригерів лічильника; $N_П$ – попередньо записане в лічильник початкове число. При лічбі має виконуватись умова

$$\sum U^- \le N_П + \sum U^+ \le 2^n - 1.$$

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 лічби використовують додатковий RS-тригер: з його прямого виходу знімається сигнал керування додаванням $Y_Д$ (вмикає кола перенесення), а з інверсного виходу – сигнал керування відніманням $Y_В$ (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.).

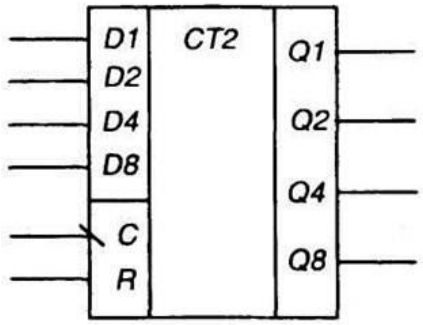



Рисунок 1.26 – Умовне позначення чотирирозрядного послідовного. двійкового лічильника

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 6;

– блокування перенесень: 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, що забезпечують лічбу в двійковому коді, та 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.

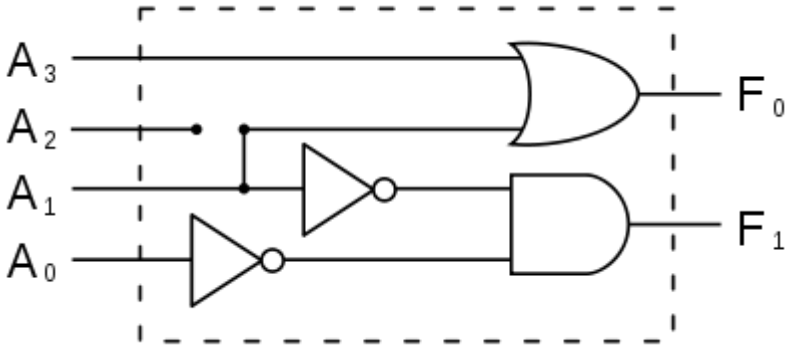



Рисунок 1.27 – Шифратор

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: $\overline{Q_0}\overline{Q_1}\ldots Q_n, \overline{Q_0}\overline{Q_1},\ldots,\overline{Q}_{n-1}\overline{Q}_n,\ldots,Q_0Q_1\ldots Q_n$. 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 незв'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].

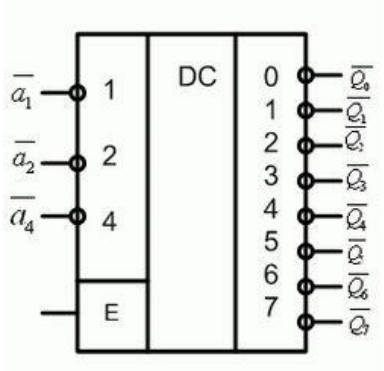



Рисунок 1.28 – Умовне позначення дешифратора 3 на 8 на схемах.

## 1.7 Мультиплексор

Мультиплексори належать 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 | $X_3$ |

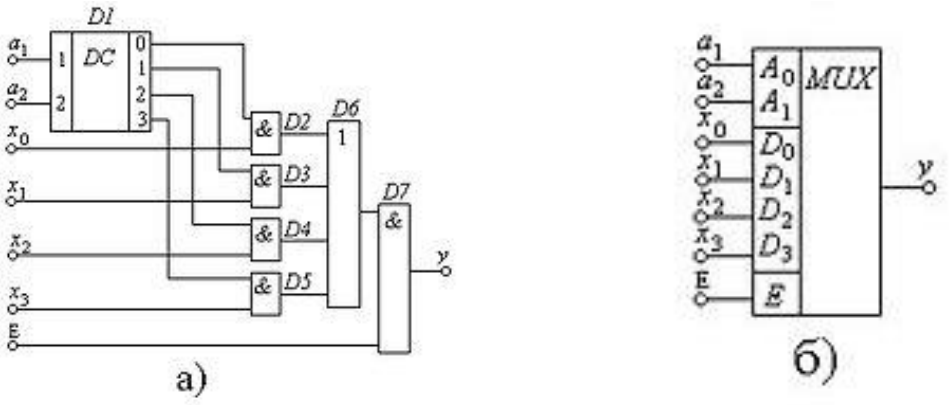



Рисунок 1.29 – Мультиплексор 4-1; а) – схема; б) – умовне позначення

## 1.8 Демультиплексор

Демультиплексор належить 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 E використовується як інформаційний. Якщо для побудови схеми демультиплексора використати дешифратор без входу дозволу E, то необхідно мати *m* двовхідних логічних елементів 2I.

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 3I 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* |

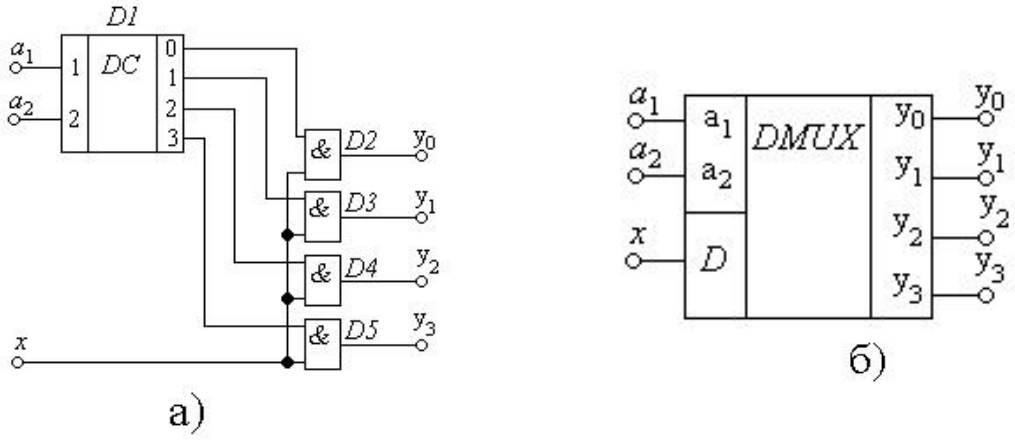



Рисунок 1.30 – Демультиплексор 1-4 на базі дешифратора D1 та логічних елементів 2I D2…D5 (без входу дозволу); а – схема; б – умовне позначення

## 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

$$F\Sigma = \frac{1}{t\Sigma},$$

тут розуміємо операції регістр-регістр (тобто числа зберігаються в регістрах арифметико-логічного пристрою);

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

– вартість багаторозрядного суматора визначається загальною кількістю використаних мікросхем;

– споживаною потужністю [9].

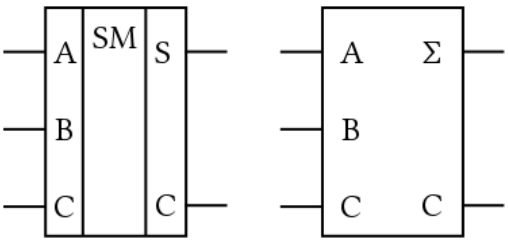



Рисунок 1.31 – Позначення суматора на електронних схемах

# ЧАСТИНА ІІ
# АРХІТЕКТУРА КОМП'ЮТЕРІВ

## 2.1 Історія розвитку обчислювальної техніки

У ході еволюції комп'ютерних технологій було розроблено сотні різних комп'ютерів. Багато з них давно забуті, в той час як вплив інших на сучасні ідеї виявився досить значним. У цьому розділі дамо короткий огляд деяких ключових історичних моментів, щоб краще зрозуміти, як розробники дійшли до концепції сучасних комп'ютерів. Зрозуміло, розглянемо лише основні моменти розвитку, залишивши багато подробиць за дужками.

**Нульове покоління – механічні комп'ютери (1642-1945).** Першою людиною, яка створила лічильну машину, був французький вчений Блез Паскаль (1623–1662) (рис. 2.1), на честь якого названо одну з мов програмування. Паскаль сконструював цю машину в 1642 році, коли йому було всього 19 років, для свого батька, збирача податків. Це була механічна конструкція з шестерінками та ручним приводом.


Рисунок 2.1 – «Паскаліна»

Рахункова машина Паскаля могла виконувати тільки операції додавання та віднімання. Через тридцять років великий німецький математик барон Готфрід Вільгельм фон Лейбніц (1646–1716) побудував іншу механічну машину (рис 2.2), яка крім додавання та віднімання могла виконувати операції множення та ділення. По суті, Лейбніц три століття тому створив кишенькову подобу калькулятора з чотирма функціями.

Рисунок 2.2 – Машина Лейбніца

Приблизно в 1820 році Чарльз Томас (Charles Xavier Thomas) створив перший вдалий механічний калькулятор, що випускався серійно – Арифмометр Томаса, який міг додавати, віднімати, множити і ділити. Механічні калькулятори, які рахували десяткові числа, використовувалися до 1970-х.

Рисунок 2.3 – Арифмометр Томаса

У 1835 році професор математики Кембриджського університету Чарльз Беббідж (1792-1871), винахідник спідометра, розробив і сконструював різницеву машину. Ця механічна машина, могла тільки додавати та віднімати, підраховувала таблиці чисел для морської навігації. У машину було закладено лише один алгоритм – метод кінцевих різниць із використанням поліномів. У цієї машини був досить цікавий спосіб виведення інформації: результати видавлювалися сталевим штампом на мідній дощечці, що передбачило пізніші засоби введення-виведення з одноразовим записом – перфокарти та компакт-диски.

Хоча його пристрій працював досить непогано, Беббіджу незабаром набридла машина, що виконувала лише один алгоритм. Він витратив дуже багато часу, більшу частину свого сімейного стану та ще 17000 фунтів, виділених урядом на розробку аналітичної машини (рис 2.4). У аналітичної машини було 4 компоненти: запам’ятовуючий пристрій (пам’ять), обчислювальний пристрій, пристрій введення (для зчитування перфокарт), пристрій виведення (перфоратор та принтер). Пам’ять складалася з 1000 слів з 50 десяткових розрядів; кожне зі слів містило змінні та результати.

Обчислювальний пристрій приймав операнди з пам’яті, потім виконував операції складання, віднімання, множення чи поділу і повертав отриманий результат у пам’ять. Як і різницева машина, цей пристрій був механічним.

Перевага аналітичної машини полягала в тому, що вона могла виконувати різні завдання. Вона зчитувала команди з перфокарт та виконувала їх.

Деякі команди наказували машині взяти два числа з пам’яті, перенести їх у обчислювальний пристрій, зробити над ними операцію (наприклад, додати) і відправити результат назад до пристрою. Інші команди перевіряли число, а іноді робили операцію переходу в залежності від того, додатне воно чи від’ємне. Якщо в пристрій для зчитування вводилися перфокарти з іншою програмою, машина виконувала інший набір операцій. Тобто на відміну від різницевої аналітична машина могла виконувати кілька алгоритмів.

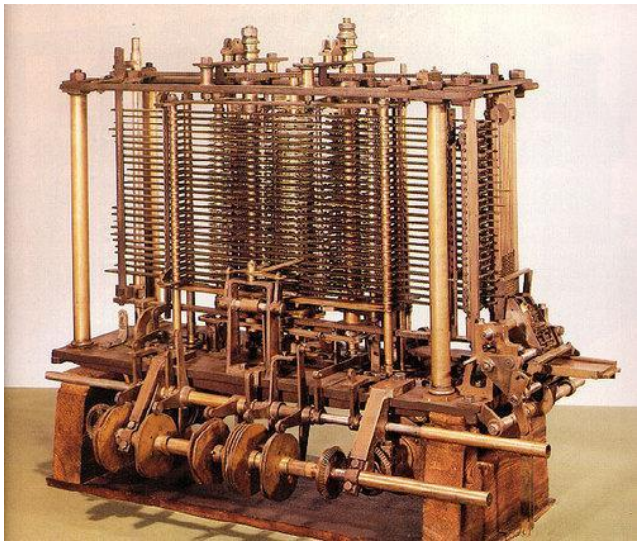


Рисунок 2.4 – Аналітична машина Чальза Беббіджа

Оскільки аналітична машина програмувалася на елементарному асемблері, їй було необхідне програмне забезпечення. Щоб створити це програмне забезпечення, Беббідж найняв молоду жінку – Аду Августу Лавлейс (Ada Augusta Lovelace), дочку знаменитого британського поета Байрона.

Ада Лавлейс була першим у світі програмістом. На її честь названо сучасний мова програмування –Ada.

На жаль, подібно до багатьох сучасних інженерів, Беббідж так ніколи і не налагодив свій комп'ютер. Йому потрібні були тисячі і тисячі шестерень, зроблених з точністю, яка у дев'ятнадцятому столітті була недоступна. Але ідеї Беббіджа випередили його епоху, і навіть сьогодні більшість сучасних комп'ютерів по конструкції подібні до аналітичної машини. Тому справедливо буде сказати, що Беббідж був дідусем сучасного цифрового комп'ютера [10].

До 1900-го року ранні механічні калькулятори, касові апарати і рахувальні машини були перепроектовані з використанням електричних двигунів з поданням положення змінної як позиції шестерні. З 1930-х такі компанії як Friden, Marchant і Monro почали випускати настільні механічні калькулятори (рис. 2.5), які могли додавати, віднімати, множити і ділити. Словом «computer» (буквально – «обчислювач») називалася посада – це були люди, які використовували калькулятори для виконання математичних обчислень.

Рисунок 2.5 – Механічний калькулятор Friden SVE

У 1948 році з’явився Curta – невеликий механічний калькулятор, який можна було тримати в одній руці.


Рисунок 2.6 – Механічний калькулятор Curta

У Радянському Союзі в той час найвідомішим і поширеним калькулятором був механічний арифмометр «Феликс», що випускався з 1929 по 1978 рік на заводах в Курську (завод «Счетмаш»), Пензі та Москві [11].

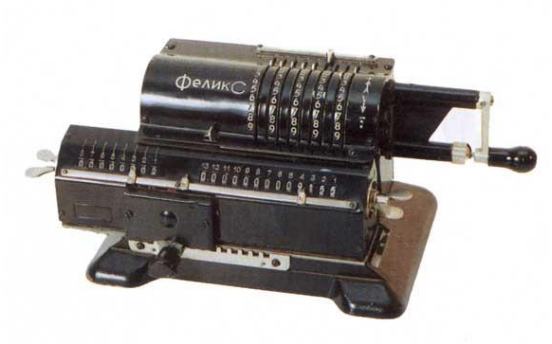



Рисунок 2.7 – Арифмометр «Феликс»

У 1936 році, працюючи в ізоляції у Німеччині, Конрад Цузе почав роботу над своїм першим обчислювачем серії Z, що мав пам’ять і (поки обмежену) можливість програмування. Z3 став першим працюючим комп’ютером, керованим програмою. У багатьох відношеннях Z3 був подібний сучасним машинам, в ньому вперше був представлений ряд нововведень, таких як арифметика з плаваючою крапкою. Заміна складної у реалізації десяткової системи на двійкову, зробила машини Цузе більш простими і більш надійними, вважається, що це одна з причин того, що Цузе досяг успіху там, де Беббідж зазнав невдачі.

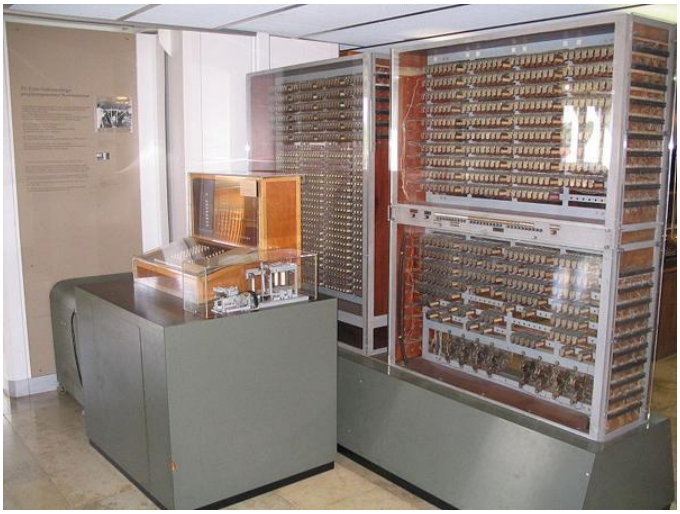


Рисунок 2.8 – Z3 Конрада Цузе

Програми для Z3 зберігалися на перфорованій плівці. Умовні переходи були відсутні, але в 1990-х було теоретично доведено, що Z3 є універсальним комп'ютером (якщо ігнорувати обмеження на розмір фізичної пам'яті).

Трохи пізніше в Америці конструюванням лічильних машин зайнялися дві людини: Джон Атанасов (John Atanasoff) з Коледжу штату Айова та Джордж Стібіц з Bell Labs. Машина Атанасова (рис 2.9) була надзвичайно потужною для того часу. У ній використовувалася двійкова арифметика та пам'ять на базі конденсаторів, які періодично оновлювалися, щоб уникнути витоку заряду.

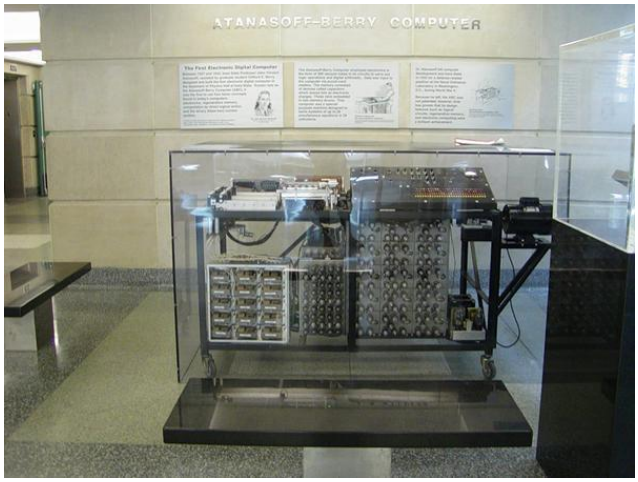



Рисунок 2.9 – Atanasoff-Berry Computer (ABC)

Сучасна динамічна пам'ять (ОЗП) працює точно за таким же принципом. На жаль, ця машина так і не стала чинною. В якомусь сенсі Атанасов був схожий на Беббіджа: провидець, мрії якого зрештою розбилися через недосконаліть технологій свого часу.

Комп'ютер Джорджа Стіббіца (George Stibbitz) дійсно працював, хоча і був примітивнішим, ніж машина Атанасова. Стіббіц продемонстрував свою машину на конференції в Дартмутському коледжі 1940 року. На цій конференції був присутній Джон Моушлі (John Mauchley), нічим не примітний на той момент

професор фізики з університету Пенсільванії. Пізніше він став дуже відомим у галузі комп'ютерних розробок.

Поки Цузе, Стіббіцс та Атанасов розробляли автоматичні рахункові машини, молодий Говард Айкен (Howard Aiken) у Гарварді вперто виконував нудні ручні обчислення для докторської дисертації. Після здобуття ступеня Айкен усвідомив важливість автоматизації обчислень. Він пішов у бібліотеку, прочитав про роботу Беббіджа і вирішив створити з реле такий самий комп'ютер, який Беббіджу не вдалося створити із зубчастих коліс.

Робота над першим комп'ютером Айкена «Mark I» (рис 2.10) була закінчена в Гарварді 1944 року. Комп'ютер мав 72 слова за 23 десятковими розрядами кожне, а час виконання операції становив 6 секунд. У пристроях введення-виводу використовувалася перфострічка. На той час, як Айкен закінчив роботу над комп'ютером «Mark II», релейні комп'ютери вже застаріли. Почалася епоха електроніки [10,11].

Рисунок 2.10 - Гарвардський Mark I

**Перше покоління – електронні лампи (1945–1955)**. Стимулом створення електронного комп'ютера стала Друга світова війна. На початку війни німецькі підводні човни завдавали серйозних збитків британському флоту. Німецькі адмірали посилали на підводні човни радіокоманди, і хоча англійці могли перехоплювати ці команди, проблема була в тому, що радіограми були закодовані за допомогою приладу під назвою ENIGMA.

На початку війни англійцям вдалося придбати ENIGMA у поляків, які, у свою чергу, вкрали її у німців. Однак, щоб розшифрувати закодоване послання, вимагалося дуже багато обчислень, та його треба було зробити відразу після перехоплення радіограми. Тому британський уряд заснувало секретну лабораторію для створення електронного комп'ютера під назвою COLOSSUS (рис 2.11). У створенні цієї машини брав участь знаменитий британський математик Алан Тьюрінг. COLOSSUS працював уже 1943 року, але оскільки британський уряд повністю контролював цей проект і розглядав його як військову таємницю протягом 30 років, COLOSSUS не став базою подальшого

розвитку комп’ютерів. Ми згадали про нього лише тому, що це був перший у світі електронний цифровий комп’ютер.

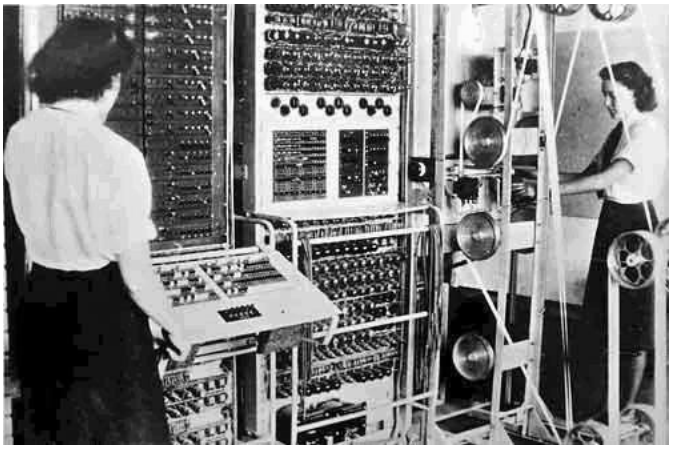


Рисунок 2.11 - Британський Colossus

Війна вплинула на розвиток комп’ютерної техніки США. Армії потрібні були таблиці, які використовувалися для наведення важкої артилерії. Сотні жінок наймалися для розрахунків на ручних рахункових машинах та заповнення полів цих таблиць (вважалося, що жінки акуратніші у розрахунках, ніж чоловіки). Тим не менш, цей процес вимагав багато часу, і в ньому часто траплялися помилки.

Джон Моушлі, який був знайомий з роботами Атанасова та Стіббіца, розумів, що армія зацікавлена у рахункових машинах. Він звернувся до армії із заявкою на фінансування робіт із створення електронного комп’ютера. Заявку було задоволено в 1943 році, і Моушлі зі своїм студентом Дж. Преспером Екертом (J. Presper Eckert) почали конструювати електронний комп’ютер, який вони назвали ENIAC (Electronic Numerical Integrator and Computer) електронний цифровий інтегратор та калькулятор). ENIAC (рис 2.12) складався з 18000 електровакуумних ламп і 1500 реле, важив 30 тон і споживав 140 кіловат електроенергії. У машини було 20 регістрів, кожен з яких міг утримувати, натискати 10-розрядне десяткове число. (Десятковий регістр – це пам’ять дуже маленького об’єму, яка може вміщувати число до якоїсь певної максимальної кількості розрядів, щось на зразок спідометра, що запам’ятовує кілометраж пройденого автомобілем шляху). Програмування ENIAC здійснювалося за допомогою 6000 багатоканальних перемикачів та численних кабелів, що підключалися до роз’ємів.

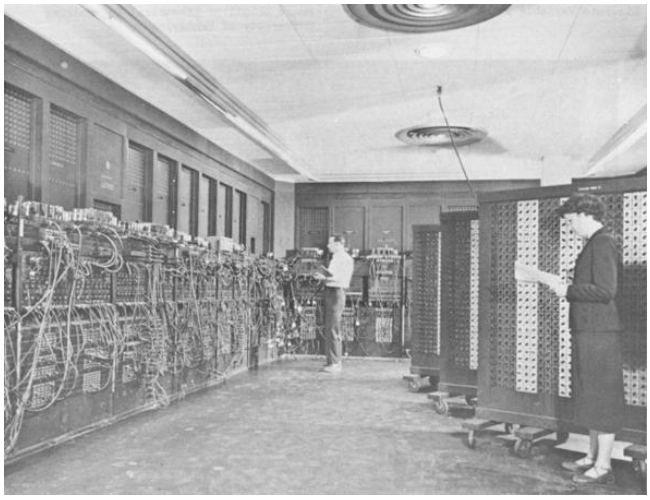


Рисунок 2.12 – ENIAC

Робота над машиною була закінчена в 1946 році, коли вона вже була не потрібною – принаймні для досягнення початкових цілей. Оскільки війна закінчилася, Моушлі та Екерту дозволили організувати школу, де вони розповідали про свою роботу колегам-науковцям. У цій школі і зародився інтерес до створення великих цифрових комп'ютерів.

Один з учасників проекту ENIAC, Джон фон Нейман, поїхав до Інституту спеціальних досліджень у Прінстоні, щоб сконструювати власну версію EDVAC під назвою IAS (Immediate Address Storage – пам'ять із прямою адресацією).

Фон Нейман знав багато мов, був фахівцем у фізиці та математиці, мав феноменальну пам'ять: він пам'ятав усе, що коли-небудь чув, бачив чи читав. Він міг дослівно процитувати текст книги, які читав кілька років назад. Коли фон Нейман став цікавитись обчислювальними машинами, він вже був найзнаменитішим математиком у світі.

Фон Нейман невдовзі зрозумів, що програмування комп'ютерів з великою кількістю перемикачів та кабелів – заняття повільне, стомлююче та незручне. Він прийшов до думки, що програма має бути представлена у пам'яті комп'ютера у цифровій формі, разом з даними. Він також зазначив, що десяткова арифметика, що використовується в машині ENIAC, де кожен розряд представлявся десятьма електронними лампами (одна включена і дев'ять вимкнено), може бути замінена паралельною двійковою арифметикою. Атанасов прийшов аналогічного висновку на кілька років раніше.

Основний проект, який фон Нейман описав спочатку, відомий зараз як фон-нейманівська обчислювальна машина.

Машина фон Неймана складалася з п'яти основних частин: пам'яті, арифметико-логічного пристрою, пристрою управління, а також пристроїв введення-виведення.

Пам'ять складалася з 4096 слів, кожне слово містило 40 біт (0 чи 1). Кожне слово містило або дві команди по 20 біт, або ціле 40-розрядне число зі знаком на 40 біт. 8 біт визначали тип команди, а решта 12 біт визначали одне із 4096 слів пам'яті. Арифметичний блок та блок управління становили «мозковий центр» комп'ютера. У сучасних машинах ці блоки поєднуються. в одній мікросхемі, яка називається центральним процесором (ЦП).

Усередині арифметико-логічного устрою знаходився особливий внутрішній регістр на 40 біт, так званий акумулятор. Типова команда додавала слово з пам'яті до акумулятора або зберігала вміст акумулятора у пам'яті [10].

**Друге покоління – транзистори (1955-1965).** Транзистор був винайдений співробітниками лабораторії Bell Laboratories Джоном Бардіном (John Bardeen), Уолтером Браттейном (Walter Brattain) та Вільямом Шоклі (William Shockley), за що в 1956 році вони отримали Нобелівську премію в галузі фізики. Протягом десяти років транзистори зробили революцію у виробництві комп'ютерів, і до кінця 50-х років комп'ютери на вакуумних лампах стали пережитком минулого. Перший комп'ютер на транзисторах був побудований у лабораторії MTI. Він містив слова з 16 біт, як і Whirlwind I.

Комп'ютер називався TX-0 (Transistorized eXperimental computer 0 – експериментальна транзисторна обчислювальна машина 0) і призначався тільки для тестування майбутньої машини TX-2.

Машина TX-2 не мала великого значення, але один з інженерів із цієї лабораторії, Кеннет Ольсен (Kenneth Olsen), у 1957 році заснував компанію DEC (Digital Equipment). Ця машина, PDP-1, з'явилася тільки через чотири роки головним чином тому, що ті, хто фінансував DEC, вважали, що виробництво комп'ютерів немає майбутнього.

Зрештою, Т. Дж. Вотсон, колишній президент IBM, одного разу сказав, що світовий ринок комп'ютерів складає чотири чи п'ять одиниць. Тому компанія DEC продавала переважно невеликі електронні плати.

Комп'ютер PDP-1 (рис 2.13) з'явився лише 1961 року. Він мав 4096 слів по 18 біт та швидкодія 200000 команд на секунду. Компанія DEC продала десятки комп'ютерів PDP-1 і так з'явилася комп'ютерна промисловість.

Рисунок 2.13 – DEC PDP-1

Одну з перших машин моделі PDP-1 віддали в MTI, де вона одразу привернула увагу деяких молодих дослідників.

Одним із нововведень PDP-1 був дисплей з розміром 512 × 512 пікселів, на якому можна було малювати крапки. Незабаром студенти MTI склали спеціальну програму для PDP-1, щоб грати у «Космічну війну» – першу у світі комп'ютерну гру.

Через кілька років компанія DEC розробила модель PDP-8, 12-розрядний комп'ютер. PDP-8 коштував набагато дешевше, ніж PDP-1 (16000 доларів).

Головне нововведення – єдина шина (omnibus). Шина – це набір паралельно з'єднаних дротів, що зв'язують компоненти комп'ютера. Це нововведення радикально відрізняло PDP-8 від IAS.

Така архітектура з того часу стала використовуватися у всіх малих комп'ютерах.

З винаходом транзисторів компанія IBM побудувала транзисторну версію IBM 709 – IBM 7090, а пізніше - IBM 7094. У цій версії час циклу становив 2 мікросекунди, а пам'ять складалася з 32536 слів по 36 біт.

У той же час, компанія IBM заробляла великі гроші на продажі невеликих комп'ютерів IBM 1401 для комерційних розрахунків. Ця машина могла зчитувати та записувати магнітні стрічки та перфокарти та роздруковувати результат так само швидко, як і IBM 7094, але при цьому коштувала дешевше. Для наукових обчислень вона не підходила, але була дуже зручна для комерційного обліку.

Архітектура IBM 1401 була незвичайною тим, що в ній не було регістрів і навіть фіксованої довжини слова. Пам'ять містила 4000 байт по 8 біт (у пізніх моделях обсяг збільшився до небачених на той час 16000 байт). Кожен байт містив символ 6 біт, адміністративний біт і біт позначення кінця слова. Команда MOVE, наприклад, використовувала дві адреси: джерела та приймача. Вона переміщала байти з джерела до приймача, доки виявляла встановлений біт кінця слова.

В 1964 маленька, нікому не відома компанія CDC (Control Data Corporation) випустила машину 6600, яка працювала майже на порядок швидше, ніж IBM 7094 та інші машини на той час. Цей комп'ютер для складних розрахунків користувався великою популярністю, і компанія CDC пішла «в гору». Секрет настільки високої швидкодії полягав у тому, що всередині ЦП (центрального процесора) знаходилася машина з високим ступенем паралелізму. У неї було кілька функціональних пристроїв для додавання, множення та ділення, і всі вони могли працювати одночасно.

Розробник комп'ютера 6600 Сеймур Крей (Seymour Cray) був легендарною особистістю, як і фон Нейман. Він присвятив своє життя створенню дуже потужних комп'ютерів, які зараз називають суперкомп'ютерами. Серед них можна назвати 6600, 7600 та Cray-1.

Слід згадати ще один комп'ютер – Burroughs B 5000 (рис 2.14). Розробники машин PDP-1, IBM 7094 і 6600 займалися лише апаратним забезпеченням, намагаючись знизити його вартість (DEC) чи змусити працювати швидше (IBM і CDC). На програмне забезпечення ніхто не звертав уваги. Виробники B 5000 пішли іншим шляхом. Вони розробили машину спеціально для програмування мовою Algol 60 (попередниця мов C та Java), сконструювавши апаратне забезпечення так, щоб спростити завдання компілятора. Так виникла ідея, що при розробці комп'ютера потрібно враховувати і програмне забезпечення. На жаль, про неї практично одразу ж забули [10, 11].

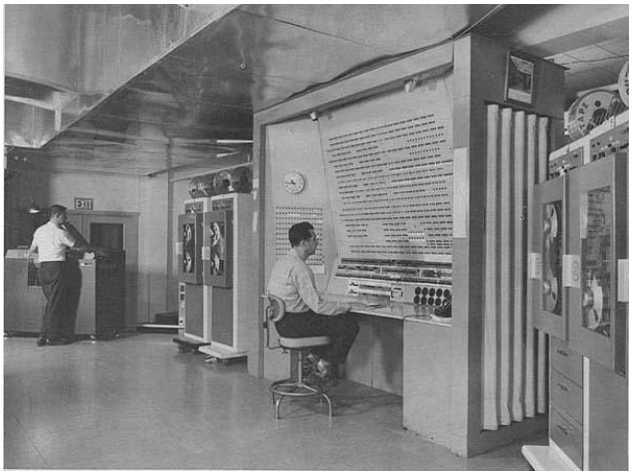


Рисунок 2.14 – Burroughs B 5000

**Третє покоління – інтегральні схеми (1965–1980).** Винахід кремнієвої інтегральної схеми (рис 2.15) в 1958 Джеком Кілбі (Jack Kilby) та Робертом Нойсом (Robert Noyce) дозволило розмістити на одній невеликій мікросхемі десятки транзисторів. Пізніше це призвело до винаходу мікропроцесора Тедом Хоффом (компанія Intel).

У конкурентній боротьбі за збільшення продаж фірми, що виробляють комп'ютери, прагнули до здешевлення і мініатюризації своєї продукції. Для цього використовувалися всі сучасні досягнення науки: пам'ять на магнітних сердечниках, транзистори, і нарешті мікросхеми. До 1965 року міні-комп'ютер PDP-8 займав обсяг порівнянний з побутовим холодильником, вартість становила близько 20 тис. доларів, крім того, спостерігалася тенденція до подальшої мініатюризації.

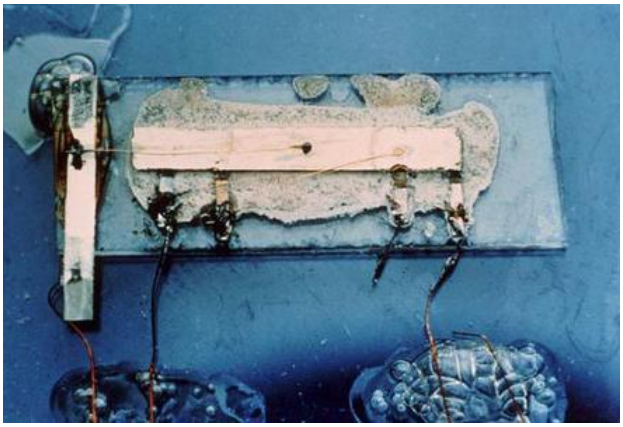


Рисунок 2.15 – Перша інтегральна схема

У 1965 р. Гордон Мур, директор підрозділу досліджень і розробок в Fairchild Semiconductor формулює висновок, заснований на спостереженнях за динамікою розвитку технологій виготовлення мікросхем. Це формулювання отримує назву закон Мура: щільність транзисторів в інтегрованих мікросхемах буде подвоюватися кожні 24 місяці протягом наступних десяти років.

4 червня 1966 року американський офіс патентів видає доктору Роберту Деннарду з компанії IBM патент № 3387286 на однотранзисторний елемент

пам’яті (DRAM Dynamic Random Access Memory – Динамічна пам’ять з довільним доступом) і на базову ідею 3-транзисторної комірки пам’яті. Такий тип пам’яті зараз використовується для короткочасного зберігання інформації.

У 1966 р. Роберт Нойс і Гордон Мур засновують корпорацію Intel. Ця компанія починає зі створення мікрочіпів пам’яті, але поступово перетворюється на компанію з виробництва мікропроцесорів.

У 1969 р. Пентагон створює чотири вузли мережі ARPAnet – прообразу сучасної Internet. День 2 вересня 1969 року вважається днем народження Інтернету.

1971 рік – винахід накопичувача на гнучкому магнітному диску, дискети діаметром в 200 мм (8"). В кінці 1970-х розміри дискет зменшилися до 133 мм (5,25") і в 1981 до 90 мм (3,5").


Рисунок 2.16 - Дискети 8", 5,25" та 3,5"

1971 рік – поява першого мікропроцесора (процесора, що міститься на інтегральній мікросхемі) Intel 4004. Цей процесор мав розрядність в 4 біта, і застосовувався, наприклад, у калькуляторах або схемах управління світлофорами. З мікропроцесорів 1970-х років, що знайшли застосування в персональних комп’ютерах, варто згадати 8-розрядні Intel 8080, MOS 6502, Motorola 6800 і 16-розрядні Intel 8086, Intel 8088.

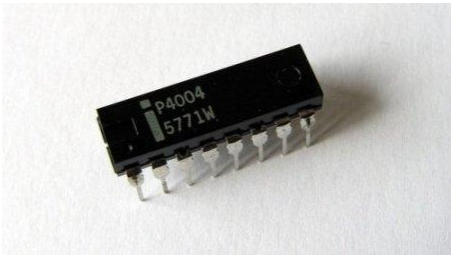



Рисунок 2.17 - Перший мікропроцесор Intel 4004

У 1972 році фірма Atari випускає першу в світі промислову телеприставку, що поклала початок епохи по-справжньому домашніх комп’ютерів, що мають звук, кольорове зображення, можливості розширення.


Рисунок 2.18 – Ігрова телеприставка Atari

У 1973 році був випущений перший прототип портативного стільникового телефону – Motorola DynaTAC. Вважається, що перший дзвінок по цьому телефону було зроблено 3 квітня 1973 року, коли його винахідник, співробітник Motorola Мартін Купер (Martin Cooper) зателефонував конкуренту з AT&T Джоелу Енгелю (Joel Engel). DynaTAC важив близько 1,15 кг і мав розмір 22,5 х 12,5 х 3,75 см. На його передній панелі було розташовано 12 клавіш, з них 10 цифрових та дві для відправки виклику і припинення розмови. У DynaTAC-а був відсутній дисплей і не було ніяких додаткових функцій. В режимі очікування він міг працювати до восьми годин, в режимі розмови близько години (за іншими даними, 35 хвилин), заряджати його доводилося трохи більше 10 годин.

Рисунок 2.19 – Motorola DynaTAC

У 1974 році фірма MITS розпочала виробництво комп’ютера Altair 8800, який, як вважається, поклав початок усім аматорським персональним комп’ютерам. Однією з причин успіху цього комп’ютера була простота архітектури по відношенню до «великих ЕОМ».

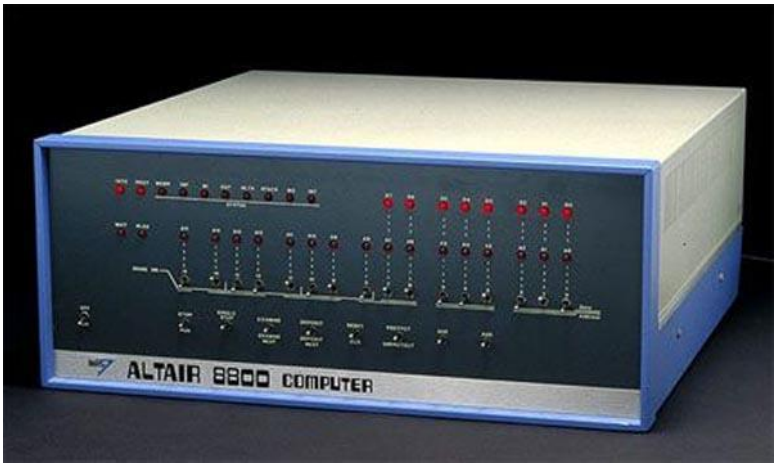



Рисунок 2.20 – Altair 8800

У 1975 р. Білл Гейтс і Пол Аллен вирішили написати інтерпретатор мови BASIC для комп'ютера Altair 8800 і заснували компанію MicroSoft, що спеціалізувалася на розробці програмного забезпечення для комп'ютерів.

У 1976 році почався кустарний випуск Apple I – комп'ютера, який послужив предтечою розвитку одного з сучасних виробників персональних комп'ютерів, Apple Inc.


Рисунок 2.21 – Apple I

У червні 1977 року перший серійний Apple II запропонував користувачам інтегровану клавіатуру, кольорову графіку, звук, пластиковий корпус і вісім слотів розширення. На відміну від усіх попередніх комп'ютерів, «Apple II» більше виглядав як офісний прилад, а не як набір електронного обладнання, мав вбудований інтерпретатор Бейсику, і був значно більш дружній до непідготовленого користувача. Тим самим «Apple II» поклав початок революції в області персональних комп'ютерів: це була машина для мас, а не тільки для аматорів, науковців або інженерів.

Рисунок 2.22 – Apple II – перший серійний персональний комп’ютер

1980 рік – перший 5,25-дюймовий Winchester, Shugart ST-506, 5 Мб [10, 11].

Рисунок 2.23 – Shugart ST-506

**Четверте покоління – надвеликі інтегральні схеми (1980-?)** Поява надвеликих інтегральних схем (HBIC) у 80-х роках дозволила розміщувати на одній платі спочатку десятки тисяч, потім сотні тисяч і, нарешті, мільйони транзисторів Це призвело до створення комп’ютерів меншого розміру та більш швидкодіючих. До появи PDP-1 комп’ютери були настільки великі та дорогі, що компаніям та університетам доводилося мати спеціальні відділи (обчислювальні центри). До 80-х років ціни настільки впали, що можливість купувати комп’ютери з’явилася не лише в організацій, а й в окремих людей. Почалася епоха персональних комп’ютерів.

Персональні комп’ютери були потрібні зовсім для інших цілей, ніж їх попередники. Вони застосовувалися для обробки слів, електронних таблиць, а також для виконання програм із високим рівнем інтерактивності (наприклад, ігор), для яких великі комп’ютери не підходили.

Перші персональні комп’ютери продавалися як комплекти. Кожен комплект містив друковану плату, набір інтегральних схем, зазвичай включаючи схему Intel 8080, кілька кабелів, джерело живлення та іноді 8-дюймовий дисковод. Скласти із цих частин комп’ютер покупець мав сам. Програмне забезпечення до комп’ютера не додається. Покупцеві доводилося писати програмне забезпечення самому. Пізніше з’явилася операційна система CP/M, написана Гарі Кілдаллом (Gary Kildall) для Intel 8080. Це була повноцінна

операційна система (на дискеті), зі своєю файловою системою. схемою та інтерпретатором для виконання користувальницьких команд, які вводилися із клавіатури.

Ще один персональний комп'ютер, Apple (а пізніше і Apple II), був розроблений Стівом Джобсом (Steve Jobs) та Стівом Возняком (Steve Wozniak). Цей комп'ютер став надзвичайно популярним серед домашніх користувачів і шкіл, що миттєво зробило компанію Apple серйозним гравцем на ринку.

Спостерігаючи за тим, чим займаються інші компанії, компанія IBM, яка лідирувала тоді на комп'ютерному ринку, теж вирішила зайнятися виробництвом персональних комп'ютерів Але замість того, щоб конструювати комп'ютер «з нуля» тільки з компонентів IBM, що зайняло б занадто багато часу, компанія зробила щось нечуване: вона надала одному зі своїх працівників, Філіпу Естриджу (Philip Estridge), велику суму грошей, наказала йому вирушити куди-небудь подалі від бюрократів головного управління компанії, що втручаються в усі процеси, що знаходиться в Армонці (шт. Нью-Йорк). Естрідж, працюючи за 2000 кілометрів від головного управління компанії, узяв за основу процесор Intel 8088 і побудував персональний комп'ютер із різноманітних компонентів.

Цей комп'ютер (IBM PC) з'явився в 1981 році і став найпопулярнішим комп'ютером в історії (рис. 2.24).

Рисунок 2.24 – IBM PC

Однак компанія IBM зробила одну незвичайну річ, про яку пізніше пошкодувала. Замість тримати проект машини в секреті (або принаймні певним чином захистити себе величезною, непроникною стіною патентів), як вона зазвичай робила, компанія опублікувала повні проекти, включаючи усі електронні схеми, у книзі вартістю 49 доларів. Ця книга була опублікована для того, щоб інші компанії могли виробляти змінні плати для IBM PC, що підвищило б сумісність та популярність цього комп'ютера. На жаль для IBM, як тільки проект IBM PC став широко відомим, багато компаній почали робити клони PC і часто продавали їх набагато дешевше, ніж IBM (бо всі складові комп'ютера можна було легко придбати). Так почалося бурхливе виробництво персональних комп'ютерів.

У 1983 році на зміну IBM PC прийшов IBM PC/XT, що включав в себе жорсткий диск.

Рисунок 2.25 - IBM PC/XT

У березні 1983 року Compaq розпочала продаж Compaq Portable – першого портативного комп’ютера, а також першого клона комп’ютерів серії IBM PC.

Рисунок 2.26 - Compaq Portable

В січні 1984 року випущено перший успішний серійний персональний комп’ютер з маніпулятором типу «миша» і повністю графічним інтерфейсом, названий Apple Macintosh, тобто перший успішний комп’ютер, який реалізував ідеї, закладені в Xerox Alto в промисловому масштабі.

Рисунок 2.27 - Apple Macintosh

У 1985 році Intel випустила перший 32-бітний процесор Intel 80386, Microsoft випускає графічну оболонку Windows 1.0.

3 квітня 1986 року з'явився в продажу перший ноутбук IBM PC Convertible від фірми IBM.


Рисунок 2.28 – IBM PC Convertible

У 1990 році Тім Бернерс-Лі розробив мову HTML (Hypertext Markup Language – мова розмітки гіпертексту; основний формат Web-документів) і прототип Всесвітньої павутини.

**П'яте покоління – комп'ютери невеликої потужності. та невидимі комп'ютери.** В 1981 року уряд Японії оголосив про наміри виділити національним компаніям 500 мільйонів доларів на розробку комп'ютерів п'ятого покоління на основі технологій штучного інтелекту, які мають потіснити «слухняні» машини четвертого покоління. Спостерігаючи за тим, як японські компанії оперативно захоплюють ринкові позиції в різних галузях промисловості - від фотоапаратів до стереосистем та телевізорів, - американські та європейські виробники в паніці кинулися вимагати у своїх урядів аналогічних субсидій та іншої підтримки.

Однак, незважаючи на великий шум, японський проект розробки комп'ютерів п'ятого покоління зрештою показав свою неспроможність і був

тихо згорнутий. У якомусь сенсі ця ситуація виявилася близькою до тієї, з якою зіткнувся Беббідж – ідея настільки випередила свій час, що для її реалізації не знайшлося адекватної технологічної бази.

Проте те, що можна назвати п'ятим поколінням комп'ютерів, все ж матеріалізувалося, але в дуже несподіваному вигляді – комп'ютери почали стрімко зменшуватись. 1989 року фірма Grid Systems випустила перший планшетний комп'ютер, який називався GridPad. Він був оснащений невеликим екраном, у якому користувач міг писати спеціальним пером. Такі системи, як GridPad, продемонстрували, що комп'ютер не повинен стояти на столі або у серверній - користувач може носити його із собою, а із сенсорним екраном і розпізнаванням рукописного тексту він стає ще зручнішим.

Модель Apple Newton, що з'явилася 1993 року, наочно довела, що комп'ютер можна вмістити в корпусі розміром із касетний плеєр. Як і GridPad, Newton використовував рукописне введення, що спочатку стало великою перешкодою на шляху до успіху. Але згодом користувальницький інтерфейс подібних машин, які тепер називаються персональними електронними секретарями (Personal Digital Assistants, PDA), або просто кишеньковими комп'ютерами, був удосконалений і набув широкої популярності. В наші дні черговим етапом їхньої еволюції стали смартфони.

Інтерфейс рукописного введення PDA був удосконалений Джеффом Хокінсом (Jeff Hawkins), який створив компанію Palm для розробки ділових кишенькових комп'ютерів, розрахованих на масового споживача. Хокінс за освітою був інженером-електротехніком, але він жваво цікавився нейробіологією (наука про людський мозок). Він зрозумів, що для підвищення надійності рукописного введення можна навчити користувачів прийомам, які спрощували сприйняття введення комп'ютером – технологія, що отримала назву «Graffiti», вимагала нетривалого навчання користувача, але зрештою підвищувала швидкість і надійність введення.

Перший кишеньковий комп'ютер Palm - Palm Pilot – мав величезний успіх, а технологія «Graffiti», яка стала одним із видатних досягнень у комп'ютерній галузі, переконливо продемонструвала можливості людського розуму щодо використання можливостей людського розуму.

Користувачі PDA любили свої пристрої, старанно використовуючи їх для управління своїм розкладом та контактами. У 1990-і роки стільникові телефони набули широкого поширення. Фірма IBM вбудувала стільниковий телефон у PDA, створивши так званий «смартфон». У першому смартфоні, який називався Simon, для введення використовувався сенсорний екран, а в розпорядженні користувача виявлялися всі можливості PDA, а також телефон, ігри та електронна пошта. Зменшення розмірів та вартості компонентів у кінцевому результаті призвело до масового поширення смартфонів. Зараз найбільшою популярністю користуються платформи Apple iPhone та Google Android.

Але навіть кишенькові комп'ютери не стали по-справжньому революційними. розробкою. Значно більшого значення надається так званим «невидимим» комп'ютерам – тим, що вбудовуються в побутову техніку, годинник, банківські картки та безліч інших пристроїв [10,11].

## 2.2 Основні поняття архітектури комп'ютерів

Сукупність пристроїв, призначених для автоматичної або автоматизованої обробки інформації називають обчислювальною технікою. Конкретний набір, пов'язаних між собою пристроїв, називають обчислювальною системою. Центральним пристроєм більшості обчислювальних систем є комп'ютер.

**Комп'ютер** – це електронний пристрій, що виконує операції введення інформації, зберігання та оброблення її за певною програмою, виведення одержаних результатів у формі, придатній для сприйняття людиною. За кожну з названих операцій відповідають спеціальні блоки комп'ютера: пристрій введення, центральний процесор, запам'ятовуючий пристрій, пристрій виведення.

Всі ці блоки складаються з окремих дрібніших пристроїв. Зокрема в центральний процесор можуть входити арифметико-логічний пристрій (АЛП), внутрішній запам'ятовуючий пристрій у вигляді регістрів процесора та внутрішньої кеш-пам'яті, керуючий пристрій (КП). Пристрій введення, як правило, теж не є однією конструктивною одиницею. Оскільки види інформації, що вводиться, різноманітні, джерел може бути декілька. Це стосується і пристрою виведення.

Схематично загальна структура комп'ютера зображена на рис. 2.29.

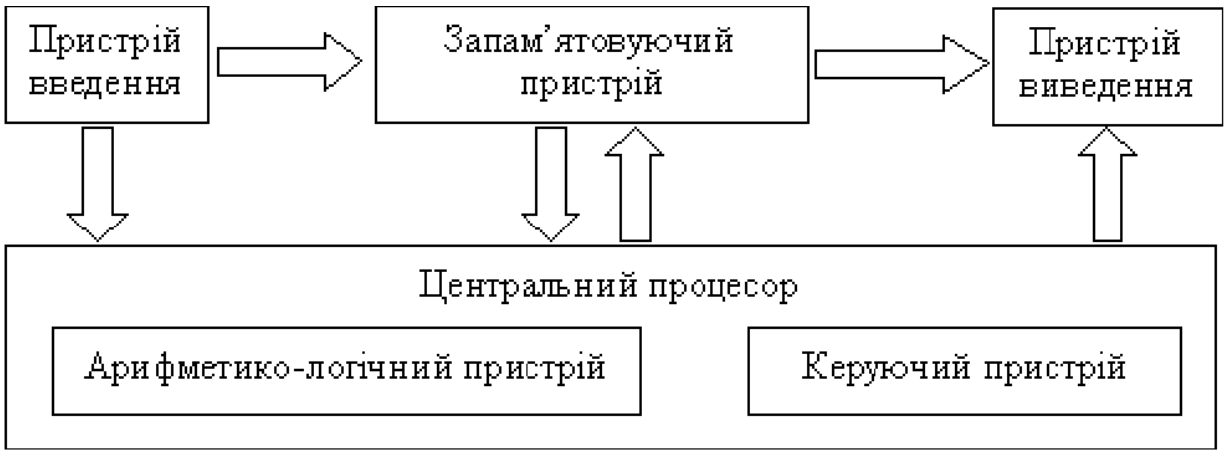



Рисунок 2.29 – Загальна структура комп'ютера

**Запам'ятовуючий пристрій** – це блок ПК, призначений для тимчасового та тривалого зберігання програм, вхідних і результуючих даних та деяких проміжних результатів. Інформація в оперативній пам'яті зберігається тимчасово лише при включеному живленні, але оперативна пам'ять має більшу швидкодію. В довготривалій пам'яті дані можуть зберігатися при вимкненому комп'ютері, проте швидкість обміну даними між даною пам'яттю та центральним процесором, у переважній більшості випадків, значно менша.

**Арифметико-логічний пристрій** – це блок ПК, в якому відбувається перетворення даних за командами програми: арифметичні дії над числами, перетворення кодів та ін.

Керуючий пристрій координує роботу всіх блоків комп'ютера. У певній послідовності він вибирає з оперативної пам'яті команду за командою. Кожна команда декодується, за потреби елементи даних з указаних в команді комірок оперативної пам'яті передаються в АЛП. АЛП настроюється на виконання дії, вказаної поточною командою (в цій дії можуть брати участь також пристрої введення-виведення); дається команда на виконання цієї дії. Цей процес буде продовжуватися доти, доки не виникне одна з наступних ситуацій: вичерпано вхідні дані, з одного з пристроїв надійшла команда на припинення роботи, вимкнено живлення комп'ютера.

Описаний принцип побудови ПК носить назву архітектури фон Неймана.

**Принципи сучасної архітектури комп'ютерів.**

**1. Принцип програмного керування.** Забезпечує автоматизацію процесу обчислень на ПК. Згідно з цим принципом, запропонованим англійським математиком Ч. Беббіджем у 1833 р., для розв'язання кожної задачі складається програма, що визначає послідовність дій комп'ютера. Ефективність програмного керування є високою тоді, коли задача розв'язується за тією самою програмою багато разів (хоч і за різних початкових даних).

**2. Принцип програми, що зберігається в пам'яті.** Згідно з цим принципом, сформульованим Дж. фон Нейманом, команди програми подаються, як і дані, у вигляді чисел й обробляються так само, як і числа, а сама програма перед виконання завантажується в оперативну пам'ять. Це прискорює процес її виконання.

**3. Принцип довільного доступу до пам'яті.** Згідно з цим принципом, елементи програм та даних можуть записуватися у довільне місце оперативної пам'яті. Довільне місце означає можливість звернутися до будь-якої заданої адреси (до конкретної ділянки пам'яті) без перегляду попередніх.

## 2.3 Методи класифікації комп'ютерів

Номенклатура видів комп'ютерів на сьогодні величезна: машини розрізняються за призначенням, потужністю, розмірами, елементною базою тощо. Тому класифікують комп'ютери за різними ознаками. Слід зауважити, що будь-яка класифікація є певною мірою умовна, оскільки розвиток комп'ютерної науки і техніки настільки стрімкий, що, наприклад, сьогоднішній ПК не поступається за потужністю серверним системам десятирічної давності. Крім того, зарахування комп'ютерів до певного класу досить умовне як через нечіткість розмежування груп, так і в наслідок впровадження в практику замовного складання комп'ютерів, коли номенклатуру вузлів і конкретні моделі їх адаптують до вимог замовника. Розглянемо найбільш поширені критерії класифікації комп'ютерів [12].

1. Класифікація за призначенням:
    а) великі комп'ютерні системи;
    б) персональні комп'ютери;
    в) промислові комп'ютерні системи.

До великих комп'ютерних системи слід віднести багатопроцесорні комп'ютерні системи об'єднані в обчислювальні кластери, на базі яких будуються Grid та Cloud технології.

Grid система - це географічно розподілена інфраструктура, яка об'єднує множину різних типів, доступ до яких користувач може отримати з будь-якої точки, незалежно від місця їх розміщення. Grid надає колективний розподілений режим доступу до ресурсів і до зв'язаних з ними послуг в рамках глобально-розподілених організацій (підприємства які спільно використовують глобальні ресурси, бази даних, спеціалізоване програмне забезпечення) [13].

Cloud технології – це модель забезпечення повсюдного та зручного доступу на вимогу через мережу до спільного пулу обчислювальних ресурсів, що підлягають налаштуванню (наприклад, до комунікаційних мереж, серверів, засобів збереження даних, прикладних програм та сервісів), і які можуть бути оперативно надані та звільнені з мінімальними управлінськими затратами та зверненнями до провайдера [14].

2. Класифікація за розміром:
   а) настільні (desktop);
   б) портативні (notebook);
   в) кишенькові (smartphone).
3. Класифікація за сумісністю:
   а) апаратна сумісність;
   б) програмна сумісність;
   в) сумісність на рівні операційної системи;
   г) сумісність на рівні даних.

Наведений перелік критеріїв класифікації є неповним.

## 2.4 Будова персонального комп'ютера

Номенклатура типів та моделей комп'ютерів достатньо велика, тому розглянемо будову персонального комп'ютера на прикладі настильного ПК. Ноутбуки, планшети та смартфони мають аналогічну будову, але у перших більшість компонентів змонтовані на одній платі, у других, частина компонентів розміщені на одному чіпі (технології SoC та SiP).

Системний блок є основною складовою настільного ПК та складається з наступних компонентів (рис. 2.30):

Материнська (головна) плата (англ. – motherboard, MB) – це основна плата, до якої приєднуються всі частини комп'ютера, встановлюється в системному блоці. Головне завдання материнської плати – об'єднати і забезпечити спільну роботу всіх інших елементів.

Центральний процесор (ЦП) – найважливіша частина ПК, що здійснює арифметико-логічні операції і керує роботою комп'ютера, встановлюється на материнську плату в спеціальний роз'єм (сокет). Саме продуктивністю процесора в першу чергу визначаються можливості комп'ютера.

Оперативна пам'ять являє собою невелику плату з розміщеними на ній мікросхемами. У них тимчасово зберігається інформація, необхідна процесору в

певний момент часу. Швидкість доступу до оперативної пам’яті досить велика. Процесор оперує такими даними, отримуючи до них майже миттєвий доступ. На материнську плату одночасно може встановлюватися кілька модулів оперативної пам’яті з метою збільшення їх загального обсягу.

Постійний запам’ятовувальний пристрій (ПЗП) – частина комп’ютера, в якій довготривало зберігається інформація (операційна система, програми, документи, тощо). На відміну від оперативної пам’яті, дані на жорсткому диску не зникають після вимикання комп’ютера або відключення ПЗП від материнської плати і живлення (дані можуть видалятися або змінюватися користувачем). У порівнянні з оперативною пам’яттю, швидкість доступу до даних на ПЗП в сотні разів нижча. ПЗП підключається до материнської плати за допомогою відповідної інформаційної шини.


Рисунок 2.30 – Основні складові системного блоку

Відеоадаптер (графічний процесор (GPU)) – частина комп’ютера, що здійснює обробку графічної інформації, встановлюється в спеціальний роз’єм материнської плати. На деяких материнських платах є вмонтовані (інтегровані) графічні процесори. В порівнянні з дискретними відеоадаптерами, продуктивність цих процесорів значно нижча, але їх можливостей цілком достатньо для вирішення нескладних завдань.

Блок живлення – обов’язковий елемент будь-якого комп’ютера, що забезпечує електроенергією всі складові частини комп’ютера.

Зовнішні носії інформації. Ці пристрої потрібні для зчитування та запису, оптичних дисків та інших типів зовнішніх накопичувачів (цей елемент не є обов'язковою частиною комп'ютера).

Корпус системного блоку – здавалося б, дуже простий елемент, але насправді від його якості, розмірів і внутрішньої структури багато в чому залежить охолодження, а відтак і термін служби складових частин комп'ютера.

Система охолодження – різного роду пристрої, що забезпечують ефективне розсіювання тепла і запобігають перегріванню окремих елементів комп'ютера. Охолодження найбільше потребують центральний процесор, відеоадаптер, схеми чіпсета материнської плати.

## 2.5 Материнська плата

Материнська плата є головною платою ПК, на якій розміщуються всі його основні елементи, лінії з'єднання та роз'єми для підключення зовнішніх пристроїв.

Тип встановленої материнської плати визначає загальну продуктивність системи, а також можливості модернізації ПК та підключення додаткових пристроїв.

На рис. 2.31 наведено приклад материнської плати, яка розроблена для процесорів Intel Core i9/Core i7/Core i5/Core i3/Pentium Gold/Celeron під сокет LGA1700.

Сокет для процесора LGA 1700 (1). Як правило на процесор встановлюється система охолодження.

Роз'єми (слоти) для встановлення модулів оперативної пам'яті (2). Зараз стандартом є 288 контактні модулі DIMM DDR4.

Роз'єми (слоти) для встановлення плат розширення (3). На даний час використовуються роз'єми шини PCI Express. Наявність слотів та можливість встановлення будь-яких карт розширення визначає відкриту архітектуру ПК.

Мікросхеми Flash пам'яті (17), в якій зберігаються програми UEFI, POST, програма початкового завантаження операційної системи, початкові налаштування (CMOS Setup).

Роз'єми для підключення накопичувачів SATA (11), M2 (6); послідовні порти для підключення периферійних пристроїв USB 3.2 (8-9), USB 2 (10), COM (14).

Роз'єми підключення живлення (5), елементів керування передньої панелі корпусу (18), вентиляторів охолодження (4)

Всі компоненти материнської плати пов'язані між собою системою провідників, яку називають інформаційною шиною.

Материнські плати характеризуються наступними параметрами.

1. Типи процесорів, які підтримує плата – визначається чипсетом. У наведеному прикладі це процесори Intel Core i9/Core i7/Core i5/Core i3/Pentium Gold/Celeron 12th Gen Intel® Core™ під сокет LGA1700.

2. Типом та кількістю оперативної пам’яті, яку підтримує плата. В нашому випадку це 4 модулі, які працюють в двоканальному режимі, загальним об’ємом до 128 Гігабайт.

3. Кількістю та типом слотів розширення. У наведеному прикладі: 1 x PCI-E 4.0 ×16 (x4), 1 × PCI-E 5.0 ×16, 2 × PCI-E 3.0 x16 (x4).

4. Кількістю та типом роз’ємів для підключення накопичувачів. В нашому випадку це 4 × SATAIII та 3 × M2.

5. Форм-фактором.

Форм-фактор – стандарт, що задає габаритні розміри технічного виробу, а також описує додаткові сукупності його технічних параметрів, наприклад форму, типи додаткових елементів, що розміщуються в пристрої, їх положення і орієнтацію [15].

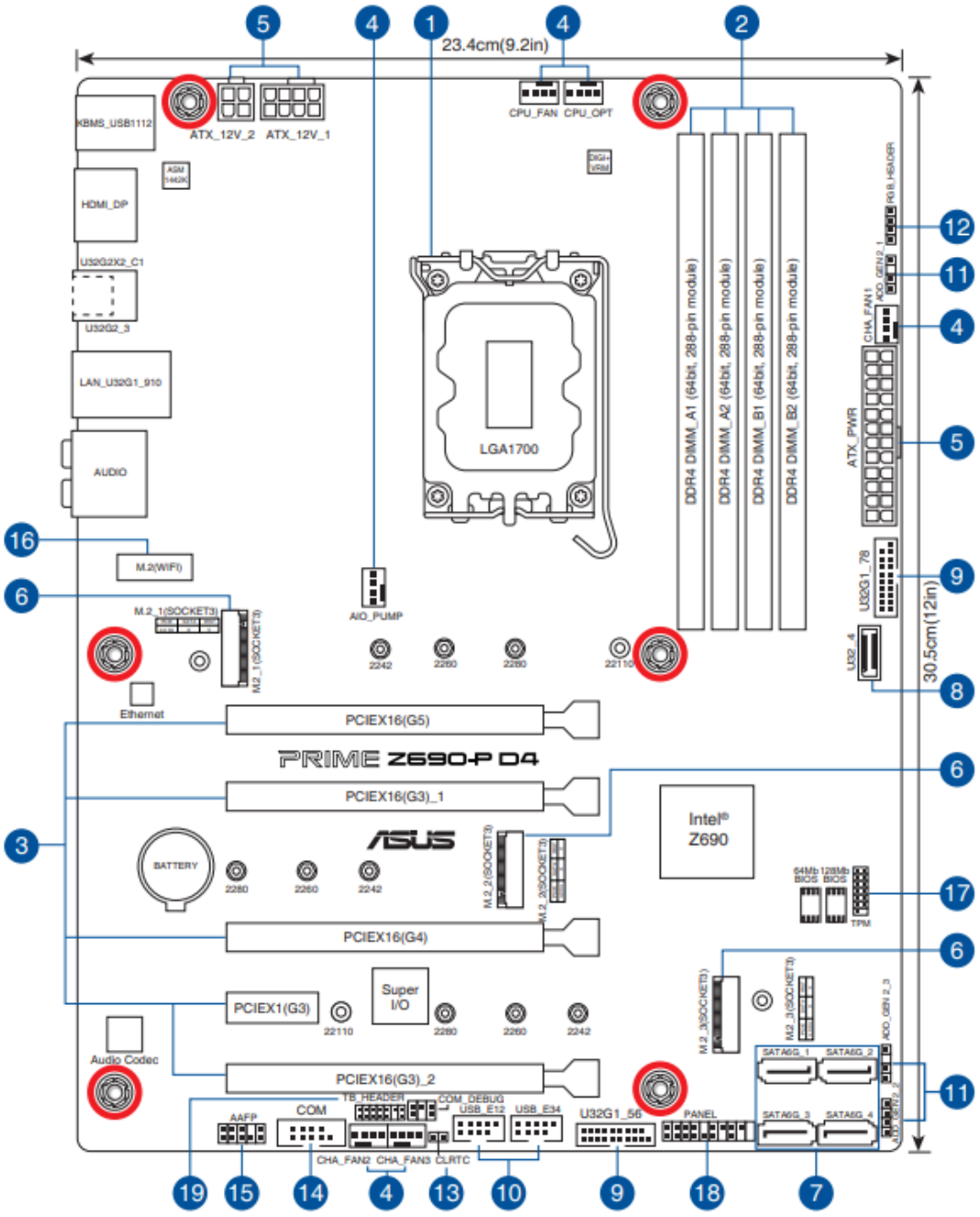



Рисунок 2.31 – Системна плата PRIME Z690-P D4 виробництва компанії Asus

Сучасні материнські плати мають наступні форм-фактори:

- ATX;
- MicroATX;
- Flex-ATX;
- NLX;
- WTX,
- CEB
- Mini-ITX;
- Nano-ITX;
- Pico-ITX;
- BTX;
- MicroBTX;
- PicoBTX.

## 2.6 Центральний процесор

Мікропроцесором у кібернетиці називають програмно-керований пристрій обробки інформації, виконаний на одній великій інтегральній мікросхемі (ВІС) або на певному наборі ВІС. З точки зору теорії цифрових пристроїв, МП – це найбільш складний на сьогоднішній день багатофункціональний цифровий пристрій, який вміщує як послідовні так і комбінаційні вузли.

За функціональним призначенням розрізняють універсальні і спеціалізовані мікропроцесори.

Універсальні МП мають алгоритмічно універсальний набір команд, за допомогою якого можна здійснювати перетворення інформації відповідно до будь-якого заданого алгоритму. Продуктивність (швидкодія) таких процесорів практично не залежить від специфіки розв’язуваних задач.

Спеціалізовані МП призначені для вирішення обмеженого і строго визначеного кола задач, іноді навіть для рішення однієї конкретної задачі. До спеціалізованих МП належать:

- сигнальні;
- медійні та мультимедійні;
- трансп’ютери;
- мікроконтролери.

Сигнальні процесори (процесори цифрових сигналів) призначені для цифрової обробки сигналів у реальному масштабі часу.

Медійні та мультимедійні процесори призначені для обробки аудіо сигналів, графічної інформації, відеозображень, а також для розв’язування ряду задач у мультимедіа комп’ютерах, ігрових приставках, побутовій техніці.

Трансп’ютери призначені для масових паралельних обчислень і роботи у мультипроцесорних системах. Для них характерним є наявність внутрішньої пам’яті та вбудованого міжпроцесорного інтерфейсу, тобто каналів зв’язку з іншими МП.

Серед спеціалізованих мікропроцесорів також можна виділити мікроконтролери – МП, призначені для рішення задач керування будь-якими процесами або пристроями [12].

Основні характеристики МП розглянемо на прикладі Intel® Core™ i5-12400F (рис 2.32-2.33), які наведені в таблиці 2.1


Рисунок 2.32 – Процесор Intel® Core™ i5-12400F

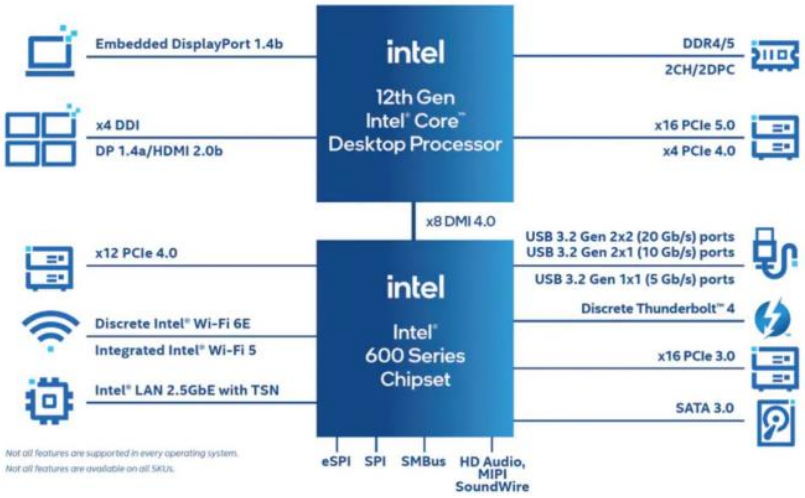



Рисунок 2.33 – Діаграма можливостей платформи 12th Gen Intel® Core™

Мікропроцесорна архітектура – це певний набір властивостей і якостей, характерних для цілого сімейства процесорів (інакше кажучи – «внутрішня конструкція», «організація» цих процесорів). В сучасних комп'ютерах (ноутбуках, планшетах, смартфонах) найрозповсюдженішими архітектурами є Intel Architecture (IA) – процесори компаній Intel, AMD та ARM – процесори компаній Apple, MediaTek, Samsung, Qualcomm та інші.

Розрядність – кількість біт, які процесор може одночасно обробляти. Фактично це розрядність регістрів загального значення МП.

Частина сучасних процесорів містять інтегрований графічний адаптер. Переміщення відеоадаптера з чипсету в процесор пов'язане з тим, що швидкість обміну даними в процесорі на порядок більша ніж швидкість обміну даними з чипсетом.

Майже всі сучасні процесори містять мінімум два обчислюваних ядра. Але для підвищення швидкодії застосовуються технології, які дозволяють виконувати на одному обчислювальному ядрі два потоки команд, наприклад

Технологія Intel® Hyper-Threading. Операційна система сприймає потік, як окремий процесор.

В сучасних процесорах для зменшення енергоспоживання тактова частота є динамічно змінюваною (це особливо важливо для ноутбуків, планшетів, смартфонів) при збільшені навантаження на МП частота збільшується, тому в характеристиках задається стандартна та максимальна тактова частота.

В основі концепції кешування (складування, від англ. cache – притулок, склад) лежить правило, що отримало назву «80/20», згідно з яким приблизно близько 20% додатків і даних ПК при обчисленні використовують близько 80% машинного часу (наприклад, ці 20% можуть включати код для надсилання та видалення електронної пошти, код збереження даних на накопичувач, код розпізнавання скан-кодів клавіатури). Тому є велика ймовірність того, що процесор знову вимагатиме ці коди і дані.

Для того щоб постачати процесор найбільш часто затребуваними даними та командами, застосовується кеш-пам’ять, яка на відміну від RAM розташована поблизу процесора і має менший час доступу.

Таблиця 2.1 – Основні характеристики Intel® Core™ i5-12400F

| Характеристика | Значення |
|---|---|
| Мікропроцесорна архітектура | IA-64 |
| Розрядність | 64 |
| Тип роз’єму | FCLGA1700 |
| Наявність інтегрованої графіки | немає |
| Кількість ядер | 6 |
| Кількість потоків | 12 |
| Тактова частота процесора | 2,5 ГГц |
| Максимальна тактова частота | 4,4 ГГц |
| Обсяг кеш пам’яті 1 рівня | 6 × 80 Кбайт |
| Обсяг кеш пам’яті 2 рівня | 6 × 1,25 Мбайт |
| Обсяг кеш пам’яті 3 рівня | 18 Мбайт |
| Кількість каналів пам’яті | 2 |
| Тип пам’яті, яку підтримує процесор | DDR4, DDR5 |
| Максимальний обсяг пам’яті | 128 Гбайт |
| Набори інструкцій які підтримує процесор | MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64E, VT-x, AES, AVX, AVX2, AVX512F, FMA3, SHA |
| Технологія віртуалізації | Intel® VT-x |
| Потужність TDP | 65Вт, максимум 117 Вт |

Сучасні процесори мають дворівневу систему кешування на кожне обчислювальне ядро (кеш L1, L2) та окремий кеш L3 загальний для всіх ядер. Кеш L1 працює за гарвардською, кеш L2, L3 за фоннейманівською архітектурами. На процесорах початкового рівня кеш L3 відсутній.

Більшість сучасних процесорів містять контролер пам’яті, який може працювати в різних режимах: одноканальному, двоканальному. Перемикання режимів задається порядком встановлення модулів пам’яті. Максимальний обсяг оперативної пам’яті залежить від розрядності шини даних процесора.

Технологія Intel®Virtualization для спрямованого введення/виводу (VT-x) дозволяє одній апаратній платформі функціонувати як кілька «віртуальних» платформ. Технологія покращує можливості управління, знижуючи час простоїв та підтримуючи продуктивність роботи за рахунок виділення окремих розділів для обчислювальних операцій.

Розширення набору команд – це додаткові інструкції, з допомогою яких можна підвищити продуктивність під час операцій із кількома об’єктами даних. До них відносяться MMX (Мультимедіа розширення), SSE (Підтримка розширень SIMD), VT-x (Технологія віртуалізації) та AVX (Векторні розширення).

Thermal design power – усереднена за часом розсіювана потужність, яку процесор, відповідно до перевірених даних, не перевищує в процесі виробництва при виконанні заданого Intel робочого навантаження високої складності на базовій тактовій частоті і при температурі спаю, вказаній в специфікації сегмента моделей і конфігурації [16].

## 2.7 Оперативна пам’ять

Принцип ієрархічної побудови пам’яті комп’ютера (запропонований Д. фон Нейманом) означає, що різні види пам’яті утворюють ієрархію, на різних рівнях якої розташовані пристрої пам’яті з різними часом доступу, складністю, вартістю та обсягом.

Необхідність ієрархічної побудови пам’яті обумовлена пошуком компромісу між великим інформаційним обсягом та високою швидкодією.

Можливість побудови ієрархії пам’яті обумовлена тим, що більшість алгоритмів в кожний проміжок часу звертаються до обмеженого набору даних, який може бути поміщений у швидшу, але дорожчу і тому невелику, за обсягом, пам’ять (рис 2.34).

Оперативна пам’ять (RAM, ОЗП) – тимчасова пам’ять, тобто дані зберігаються у ній лише до вимкнення ПК. Конструктивно оперативна пам’ять ПК виконується як модулі, отже за бажання можна порівняно просто замінити їх або встановити додаткові і цим змінити (скоріше всього, збільшити) обсяг оперативної пам’яті ПК.

За технологією зберігання ОЗП поділяють на статичну та динамічну. В статичній пам’яті в якості комірки пам’яті використовується тригер. Застосовується статична пам’ять в якості кеш-пам’яті процесору.

В динамічній пам’яті конденсатору відводиться роль безпосереднього зберігача інформації, обсяг якого складає всього один біт. Відсутність заряду на обкладинках відповідає логічному нулю, а його наявність - логічній одиниці. Транзистор грає роль «ключа», що утримує конденсатор від розряду. У спокійному стані транзистор закритий, але варто подати на відповідний рядок

матриці електричний сигнал, він відкриється, з'єднуючи обкладку конденсатора з відповідним їй стовпцем (рис. 2.35).

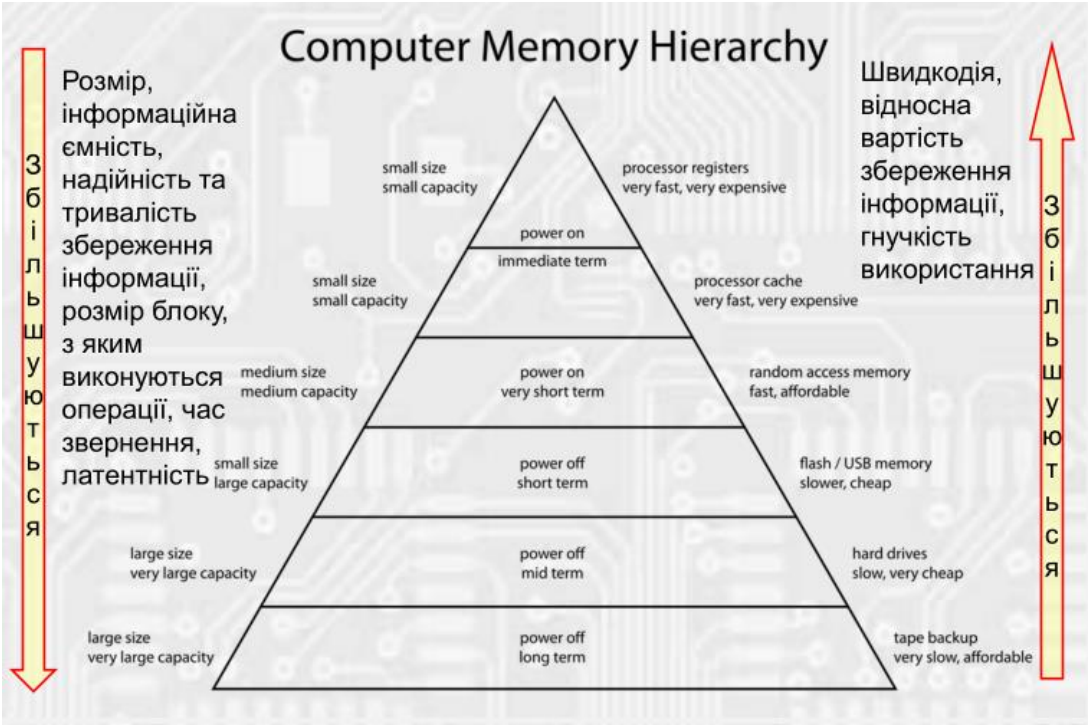



Рисунок 2.34 – Ієрархічна структура пам'яті комп'ютера

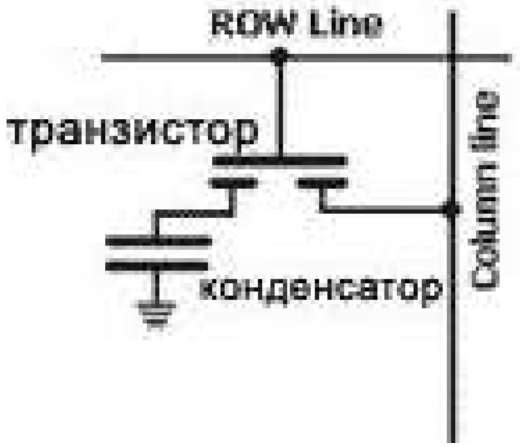



Рисунок 2.35 – Будова одного вузла динамічної оперативної пам'яті

**Типи ОЗП та їх продуктивність.** У питаннях продуктивності пам'яті спостерігається деяка плутанина, оскільки зазвичай вона вимірюється в наносекундах, в той час як швидкодія процесорів - у мегагерцах і гігагерцах. У нових швидкодіючих модулях пам'яті швидкодія вимірюється в мегагерцах, що додатково ускладнює ситуацію. На щастя, перевести одні одиниці вимірювань в інші не складає труднощів.

Наносекунда – це одна мільярдна частка секунди, тобто дуже короткий проміжок часу. Зокрема, швидкість світла у вакуумі дорівнює 299 792 км/с, тобто

за одну мільярдну частку секунди світловий промінь проходить відстань, рівну всього 29,98 см, тобто меншу за довжину звичайної лінійки.

Швидкодія мікросхем пам’яті та систем в цілому виражається в мегагерцах (МГц), тобто в мільйонах тактів на секунду, або ж у гігагерцах (ГГц), тобто в мільярдах тактів на секунду. Сучасні процесори мають тактову частоту від 2 до 5 ГГц, хоча набагато більший вплив на їх продуктивність надає їх внутрішня архітектура (наприклад, багатоядерність).

Як можна помітити, при збільшенні тактової частоти тривалість циклу зменшується.

У ході еволюції комп’ютерів для підвищення ефективності звернення до пам’яті створювалися різні рівні кешування, які дозволяють перехоплювати звернення процесора до більш повільної основної пам’яті [12].

На даний час в якості оперативної пам’яті використовується DDR SDRAM.

DDR SDRAM (від англ. Double Data Rate Synchronous Dynamic Random Access Memory – синхронна динамічна пам’ять з довільним доступом і подвійною швидкістю передачі даних) – тип комп’ютерної пам’яті, який використовується у обчислювальній техніці в якості оперативної та відеопам’яті.

При використанні DDR SDRAM досягається подвійна швидкість роботи, не бажана в SDRAM, для розрахунку команд і даних не тільки на фронті, як в SDRAM, але і по спаду тактового сигналу. За рахунок цього подвоюється швидкість передачі даних без збільшення частоти тактового сигналу шини пам’яті. Таким чином, при роботі DDR на частоті 133 МГц ми отримаємо ефективну частоту 266 МГц (при зіставленні з аналогом SDR SDRAM). У специфікації JEDEC є зауваження, що використовувати термін «МГц» в DDR некоректно, правильно показувати швидкість «мільйонів передач у секунду через один висновок даних». В таблиці 2.2 наведено основні стандарти та характеристики DDR SDRAM

Для кожного типу DDR пам’яті використовується свій тип модуля DIMM та SODIMM, які відрізняються кількістю контактів та ключем (рис 2.36).

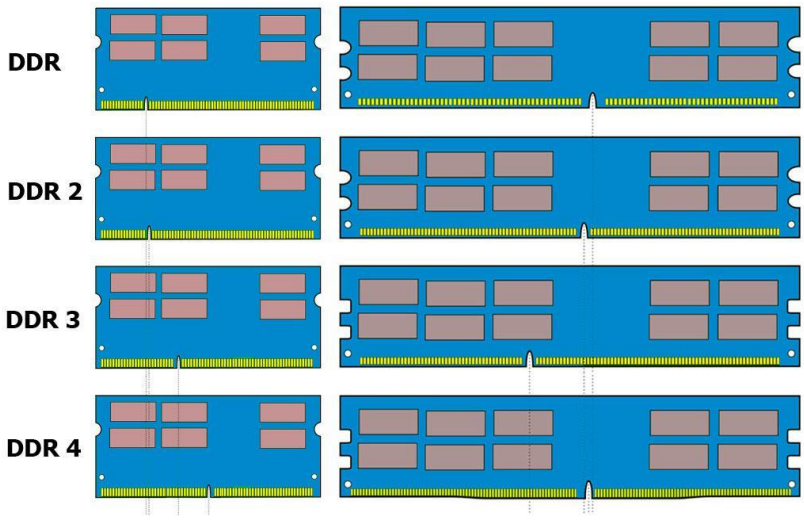



Рисунок 2.36 – Модулі DIMM та SODIMM

Таблиця 2.2 – Тактові частоти, пропускна здатність і назви модулів DDR SDRAM

| Standard | I/O clock rate | M transfers/s | DRAM name | MiB/s/DIMM | DIMM name |
|---|---|---|---|---|---|
| DDR1 | 133 | 266 | DDR266 | 2128 | PC2100 |
| DDR1 | 150 | 300 | DDR300 | 2400 | PC2400 |
| DDR1 | 200 | 400 | DDR400 | 3200 | PC3200 |
| DDR2 | 266 | 533 | DDR2-533 | 4264 | PC4300 |
| DDR2 | 333 | 667 | DDR2-667 | 5336 | PC5300 |
| DDR2 | 400 | 800 | DDR2-800 | 6400 | PC6400 |
| DDR3 | 533 | 1066 | DDR3-1066 | 8528 | PC8500 |
| DDR3 | 666 | 1333 | DDR3-1333 | 10664 | PC10700 |
| DDR3 | 800 | 1600 | DDR3-1600 | 12800 | PC12800 |
| DDR4 | 1333 | 2666 | DDR4-2666 | 21300 | PC21300 |
| DDR4 | 1600 | 3200 | DDR4-3200 | 25600 | PC25600 |
| DDR4 | 1800 | 3600 | DDR4-3600 | 28800 | PC28800 |
| DDR5 | 2600 | 5200 | DDR5-5200 | 41600 | PC41600 |
| DDR5 | 2800 | 5600 | DDR5-5600 | 44800 | PC44800 |
| DDR5 | 3000 | 6000 | DDR5-6000 | 44800 | PC48000 |

## 2.8 Накопичувачі інформації

Для зберігання інформації в сучасних комп’ютерах використовується дві основних технологій: магнітна та електронна.

Магнітна технологія використовується в накопичувачах на жорстких магнітних дисках (вінчестер, Hard Disk Drive). Жорсткий диск – це пристрій для довготривалого збереження великих об’ємів даних та програм. Ззовні, вінчестер являє собою плоску герметично закриту коробку, всередині якої знаходяться на спільній осі декілька жорстких алюмінієвих або скляних пластинок круглої форми. Поверхня кожного з дисків покрита тонким феромагнітним шаром (речовини, що реагує на зовнішнє магнітне поле), власне на ньому зберігаються записані дані. При цьому запис проводиться на обидві поверхні кожної пластини за допомогою блоку спеціальних магнітних головок. Кожна головка знаходиться над робочою поверхнею диска на відстані 0,5-0,13 мкм. Пакет дисків обертається безперервно і з великою частотою (5400-7200 об/хв.), тому механічний контакт головок і дисків недопустимий (рис. 2.37).

Запис даних у жорсткому диску здійснюється наступним чином. При зміні сили струму, що проходить через головку, відбувається зміна напруженості динамічного магнітного поля в щілині між поверхнею та головкою, що приводить до зміни стаціонарного магнітного поля феромагнітних частин покриття диску. Операція зчитування відбувається у зворотному порядку. Намагнічені частинки феромагнітного покриття спричиняють електрорушійну силу самоіндукції магнітної головки.

Електромагнітні сигнали, що виникають при цьому, підсилюються й передаються на обробку.


Рисунок 2.37 – Будова жорсткого диску

Роботою вінчестера керує спеціальний апаратно-логічний пристрій - контролер жорсткого диска [12].

Жорсткі диски відрізняються форм фактором (рис 2.38).


Рисунок 2.38 – Форм-фактор жорстких дисків

Електронна технологія збереження інформації використовується в твердотілих накопичувачах.

Твердотілий накопичувач (англ. SSD, solid-state drive) – комп'ютерний запам'ятовувальний пристрій на основі мікросхем пам'яті та контролера керування ними, що не містить рухомих механічних частин.

Розрізняють два види твердотілих накопичувачів: SSD на основі динамічної пам'яті (подібної до оперативної пам'яті комп'ютерів), і SSD на основі флеш-пам'яті.

Переважна більшість SSD-накопичувачів, представлених на ринку, засновані на флеш-пам'яті типу NAND. В залежності від того, скільки бітів даних (від одного до п'яти) можна зберегти в комірці, вони, відповідно, поділяються на такі типи: SLC, MLC, TLC, QLC пам'ять (вже випускаються) і PLC (знаходиться в розробці). Коли комірки розміщені в одній площині, говорять про планарну

флеш-пам’ять або 2D NAND, а коли вони зібрані в тривимірну структуру – про 3D NAND (окремі її різновиди називаються V-NAND або BiSC).

Будь-яка комірка пам’яті, незалежно від типу, – SLC, MLC, TLC або QLC пам’ять – це мікроскопічний транзистор. Суть роботи кожного транзистора (їх різновидів дуже багато) в тому, що електрика проходить через нього, між стоком/колектором і витоком/емітером, лише при певних умовах, регульованих поданням керуючого струму на затвор/базу. Для флеш-пам’яті використовуються польові транзистори з двома затворами: звичайним (керуючим) і спеціальним (плаваючим).

Затвори в цих транзисторах відокремлені шаром діелектрика; при подачі на затвор напруги той генерує електромагнітне поле, що впливає на середовище між стоком і витоком. Коли на затворі немає напруги, струм не проходить через транзистор, є - проходить.

На відміну від звичайного, другий, плаваючий затвор оточений діелектриком і не має виходів назовні. Таким чином:

- якщо записати в плаваючий затвор заряд, то він збережеться там навіть при знеструмленні носія (SSD або флешки);
- якщо заряд протилежний тому, що подається на керуючий затвор, ці заряди починають взаємно компенсувати один одного, і, в результаті, напруга подана на затвор, а струм через транзистор не протікає (до тих пір, поки напруга не перевищить деяке граничне значення).

Тобто можна зберігати стан комірки – той самий біт: «є заряд» або «немає заряду», поки SSD знеструмлено; і в будь-який момент подаючи напругу на затвор і перевіривши, чи йде струм між витоком і стоком, прочитати його значення.

Типи NAND-пам’яті.

- SLC («Single Level Cell», однорівнева комірка) – найстаріший тип пам’яті NAND, де на кожну комірку припадає по одному біту інформації (в комірці зберігається одне з двох значень – «0» або «1»).
- MLC («Multi Level Cell», багаторівнева комірка) – містить два біти інформації. Кількість можливих значень в такій комірці зростає до чотирьох – «00», «01», «10», «11», а порогових напруг необхідно три.
- TLC («Triple Level Cell», трирівнева комірка) – тип комірок пам’яті, SSD, що містить три біта інформації. Число значень знову збільшилася вдвічі: «000», «001», «010», «011», «100», «101», «110» і «111». Число порогових напруг – сім.
- QLC («Single Level Cell», чотирирівнева комірка), що містить, відповідно, чотири біта. Тут кількість можливих значень вже дорівнює 16: «0000», «0001», «0010», «0011», «0100», «0101», «0110», «0111», «1000», «1001», «1010», «1011», «1100», «1101», «1110», «1111» – і вимагає 15 порогових напруг.
- PLC («Penta Level Cell», п’ятирівнева комірка). Цей тип флеш пам’яті SSD тільки проанонсований до випуску і буде містити 5 бітів на клітинку, кількість можливих значень – 32: «00000», «00001», «00010», «00011», «00100», «00101», «00110», «00111», «01000», «01001», «01010», «01011», «01100», «01101», «01110», «01111», «10000», «10001», «10010», «10011», «10100»,

«10101», «10110», «10111», «11000», «11001», «11010», «11011», «11100», «11101», «11110», «11111»; порогових напруг – 31 (рис 2.39).

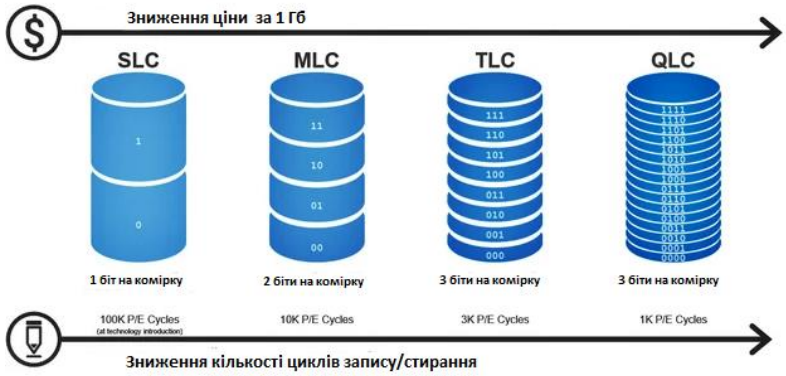



Рисунок 2.39 – Залежності ціни та кількості циклів запису від типу NAND пам’яті

3D-NAND - перехід від двовимірного, планарного масиву комірок до тривимірного розміщення та підвищенню ємності за рахунок збільшення висоти кристала. Піонером цього методу виявилась компанія Samsung з 24-шаровим, а потім 32-х і 48-шаровим V-NAND (Vertical NAND). Компанії Toshiba (нині – Kioxia) і SanDisk, які об’єдналися для спільної розробки тривимірного продукту BiCS 3D NAND, вважали за краще не форсувати події, а дочекатися, поки виробництво 3D-пам’яті стане рентабельним, і вийшли на масовий ринок вже на етапі 48- і 64-шарових чіпів (рис 2.40) [17].

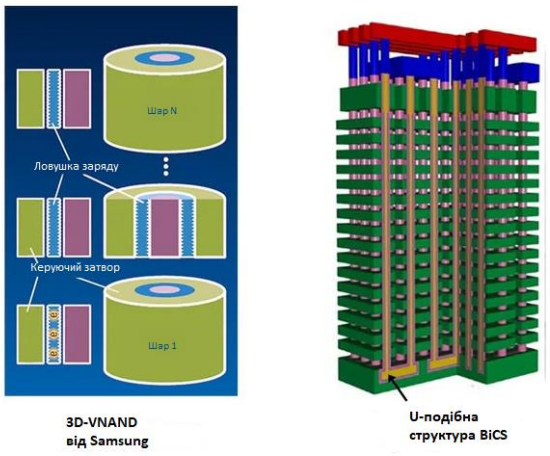



Рисунок 2.40 – Структура 3V-NAND та BiCS пам’яті

Твердотілі накопичувачі використовуються в компактних пристроях: ноутбуках, нетбуках, комунікаторах і смартфонах (рис 2.41).

Існують і так звані гібридні жорсткі диски, що з’явилися, в тому числі, через пропорційно вищу вартість твердотілих накопичувачів, поєднують в одному пристрої як основний накопичувач на твердих магнітних дисках (HDD), так і твердотілий накопичувач відносно невеликого обсягу (4-8 ГБ) в ролі кешу

(для збільшення продуктивності, швидкого холодного запуску системи, зниження енергоспоживання).

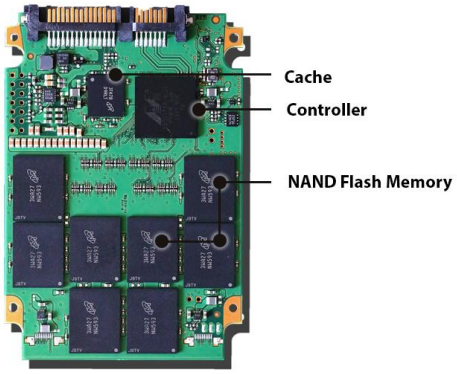



Рисунок 2.41 – Будова твердотілого диску

## Інтерфейси підключення накопичувачів

**SATA** (англ. Serial ATA) – послідовний інтерфейс обміну даними з накопичувачами інформації. SATA є розвитком інтерфейсу ATA (IDE), який після появи SATA був перейменований в PATA (Parallel ATA).

На початку стандарт SATA передбачав роботу шини на частоті 1,5 ГГц, що забезпечує пропускну здатність приблизно в 1,2 Гбіт/с (150 МБ/с). (20%-а втрата продуктивності пояснюється використанням системи кодування 8B/10B, при якій на кожні 8 біт корисної інформації припадає 2 службових біта). Пропускна здатність SATA/150 незначно вище пропускної здатності шини Ultra ATA (UDMA/133).

Стандарт SATA/300 працює на частоті 3 ГГЦ, забезпечує пропускну здатність до 2,4 Гбіт/с (300 МБ/с). Уперше був реалізований у контролері чипсету nForce 4 фірми NVIDIA. Досить часто стандарт SATA/300 називають SATA II.


Рисунок 2.42 – Роз’єм SATA

Serial ATA International Organization (SATA-IO), відповідальна за розвиток послідовного інтерфейсу, в травні 2009 року опублікувала специфікації

стандарту SATA 3.0, здатного передавати дані на швидкості до 6 Гбіт/с, удвічі вище в порівнянні з SATA 2.

SATA використовує 7-контактний роз'єм (рис 2.42). SATA-кабель має меншу ширину, за рахунок чого зменшується опір повітрю, що обдуває компоненти комп'ютера; поліпшується охолодження системи.

SATA-кабель за рахунок своєї форми стійкіший до багаторазового підключення. Шнур живлення SATA так само розроблений з урахуванням багаторазових підключень. Роз'єм живлення SATA подає 3 напруги: +12 В, +5 В і +3,3 В, однак сучасні пристрої можуть працювати без напруги +3,3 В, що дає можливість використати пасивний перехідник зі стандартного роз'єму IDE на SATA. Ряд пристроїв SATA постачається з двома роз'ємами живлення: SATA й Molex [18].

**M.2** (раніше відомий як Next Generation Form Factor і NGFF) – специфікація компактних комп'ютерних карт розширення та їхніх роз'ємів. Був створений для заміни формату mSATA і Mini PCI-E, який використовував фізичний роз'єм і розміри модулів PCI Express Mini Card. Стандарт M.2 допускає більш різноманітні розміри модулів, як по ширині, так і по довжині (рис 2.43). Формат M.2 часто використовується для реалізації продуктивних твердотілих накопичувачів на базі флеш-пам'яті, SSD), особливо при використанні в компактних пристроях, таких як ультрабуки і планшети.

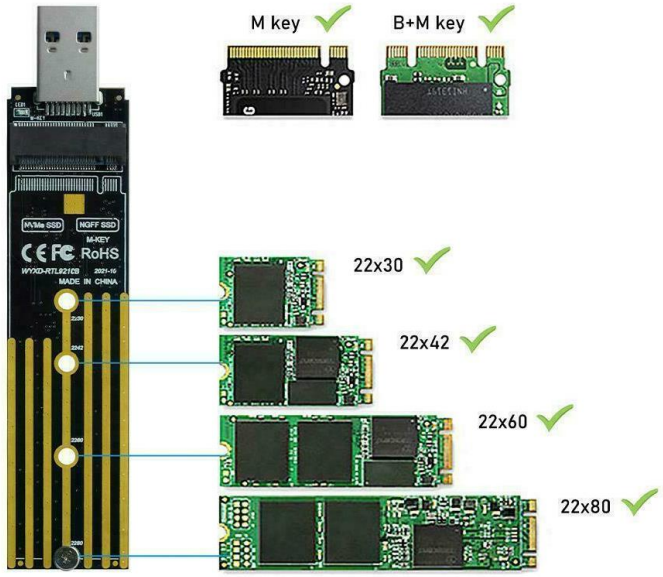



Рисунок 2.43 – Форм-фактор карт M.2

Інтерфейси, виведені на роз'єм M.2 є надмножиною інтерфейсу SATA Express. Фактично, M.2 є більш компактною реалізацією SATA Express (надає підтримку шин PCI Express 3.0 і SATA 3.0), доповненої внутрішнім інтерфейсом USB 3.0. Плати M.2 можуть мати різні ключові вирізи для позначення конкретного варіанту інтерфейсу, що використовується [19].

**Serial Attached SCSI (SAS)** – комп'ютерний інтерфейс, розроблений для обміну даними з такими пристроями як жорсткі диски, накопичувачі на оптичному диску тощо. SAS використовує послідовний інтерфейс для роботи з

безпосередньо підключеними накопичувачами (англ. Direct Attached Storage (DAS) devices). SAS розроблений для заміни паралельного інтерфейсу SCSI і дозволяє досягти більш високої пропускної здатності, ніж SCSI, в той же час SAS сумісний з інтерфейсом SATA. Хоча SAS використовує послідовний інтерфейс на відміну від паралельного інтерфейсу, що використовуються традиційним SCSI, для управління SAS-пристроями, як і раніше використовуються команди SCSI. Протокол SAS розроблений і підтримується комітетом T10. Поточну робочу версію специфікації SAS можна завантажити з його сайту. SAS підтримує передачу інформації зі швидкістю до 12 Гбіт/с. Завдяки зменшеному роз'єму SAS забезпечує повне двохпортове підключення як для 3,5-дюймових, так і для 2,5-дюймових дискових накопичувачів (раніше ця функція була доступна тільки для 3,5-дюймових дискових накопичувачів з інтерфейсом Fibre Channel) [20].

## 2.9 Відеоадаптер

Відеоадаптер (графічний адаптер, графічна карта) – важлива і дуже складна складова частина комп'ютера. Сучасні відеокарти є свого роду спеціалізованими комп'ютерами, що складаються з власного процесора, оперативної пам'яті, BIOS і інших компонентів, які за своєю структурою і організацією взаємодії пристосовані для максимально ефективного вирішення одного завдання – обробки і формування графічних даних, а також їх виведення на монітор (рис 2.44).


Рисунок 2.44 – Сучасний дискретний відеоадаптер

Сучасний відеоадаптер складається з наступних частин:

- графічний процесор (Graphics processing unit) – графічний процесорний пристрій, що займається розрахунками та формуванням графічної інформації, яка виводиться на монітор, є основою відеокарти і за своєю складністю практично не поступається центральному процесору комп'ютера, а іноді і перевершує його;
- відеопам'ять – виконує роль своєрідного буфера, в який тимчасово поміщаються зображення, що виводяться на монітор, створюються та постійно змінюються графічним ядром. У цей буфер поміщаються також елементи, необхідні для формування цих зображень;

- відеоконтролер – відповідає за правильне формування і передачу потрібної інформації з відеопам’яті на цифро-аналоговий перетворювач.
- цифро-аналоговий перетворювач (Random Access Memory Digital-to-Analog Converter) – пристрій, що здійснює перетворення цифрових результатів роботи відеокарти в аналоговий сигнал, який відображається на моніторі. Можливостями цього пристрою визначається кількість відображуваних кольорів, насиченість картинки та ін. Цифрові монітори, проектори та інші пристрої, які підключаються до цифрових роз’ємів відеокарти, використовують власні цифро-аналогові перетворювачі і від цифро-аналогового перетворювача відеокарти не залежать. Цей модуль присутній у відеоадаптерів, які підтримують аналоговий інтерфейс (VGA);
- відео-ПЗП (Video ROM) – мікросхема, що містить в собі базову систему введення-виведення відеокарти, а інакше кажучи її BIOS – сукупність правил і алгоритмів, визначених виробником, за якими складові частини відеокарти працюють і взаємодіють між собою;
- система охолодження – пристрій, що здійснює відвід і розсіювання тепла від відеопроцесора, відеопам’яті та інших компонентів графічної плати з метою забезпечення нормального температурного режиму їх роботи [10].

Основні технічні показники відеоадаптера на прикладі відеоадаптера Asus PCI-Ex GeForce GTX 1660 Super TUF Gaming OC Edition 6GB GDDR6 (рис 2.44).

1. Продуктивність відеопам’яті. Як свідчить практика, відеопам’ять дуже часто є слабким місцем графічних плат. І справа в першу чергу не в її обсязі, а в пропускній здатності, що визначає швидкість доступу до даних, які в ній зберігаються. Пропускна здатність залежить від двох показників – частоти роботи пам’яті (1800-21000 МГц) і ширини шини пам’яті (64-4096 біт) – кількості даних, що передаються за один такт. В нашому випадку частота пам’яті складає 14002 МГц, розрядність 192 біта.

2. Тип відеопам’яті (GDDR2, GDDR3, GDDR4, GDDR5, GDDR6 та ін.) вказує на те, до якого покоління належить пам’ять графічної карти. Кожне наступне покоління є досконаліше попереднього і забезпечує більш високу частоту роботи. В нашому прикладі це пам’ять GDDR6.

3. Об’єм відеопам’яті (від 1 до 48 Гб) також впливає на продуктивність графічної плати, але тільки до певної межі (коли він є слабким місцем). Набагато вигідніше придбати карту з пам’яттю GDDR3 – 256 біт і об’ємом 512 МБ ніж з пам’яттю GDDR3 – 128 біт і об’ємом 1 ГБ. В нашому прикладі це пам’ять 6 Гб.

4. Характеристики графічного ядра. Тактова частота графічного процесора є важливою, але не найголовнішою його характеристикою. Графічне ядро з порівняно невисокою частотою нерідко виявляється дуже продуктивним. Все залежить від архітектури графічного ядра. На практиці, чим новіша лінія відеокарт, до якої належить графічний прискорювач, тим, як правило, він потужніший. Виняток становлять «молодші» моделі лінії. Нерідко вони виявляються менш продуктивними, ніж «старші» представники попередньої лінії. В нашому прикладі це GPU GeForce GTX 1660 Super з тактовими частотами від 1530 до 1845 МГц.

5. Інтерфейс підключення до материнської плати. Стандартним інтерфейсом для підключення відеокарт в даний час є шина PCI-Express. Послідовна передача даних в режимі «точка-точка», застосована в PCI-Express, забезпечує можливість її масштабування (у специфікаціях описуються реалізації PCI-Express 1x, 2x, 4x, 8x, 16x і 32x) різних версій. Пропускна спроможність, з урахуванням двонаправленої передачі, для шин PCI Express з різною кількістю зв'язків вказана в таблиці 2.3.

Таблиця 2.3 – Пропускна здатність шини PCI Express

| Версія інтерфейсу | 1x | 2x | 4x | 8x | 12x | 16x | 32x |
|---|---|---|---|---|---|---|---|
| PCI Express 1.0, Гб/с | 0,5 | 1 | 2 | 4 | 6 | 8 | 16 |
| PCI Express 2.0, Гб/с | 1 | 2 | 4 | 8 | 12 | 16 | 32 |
| PCI Express 3.0, Гб/с | 2 | 4 | 8 | 16 | 24 | 32 | 64 |
| PCI Express 4.0, Гб/с | 4 | 8 | 16 | 32 | 48 | 64 | 128 |
| PCI Express 5.0, Гб/с | 8 | 16 | 32 | 64 | 96 | 128 | 256 |
| PCI Express 6.0, Гб/с | 16 | 32 | 64 | 128 | 192 | 256 | 512 |

6. Інтерфейс підключення до монітору. Підключити монітор до комп'ютера можна за допомогою таких інтерфейсів: VGA, DVI, HDMI та Display Port.

**VGA (D-Sub)** – 15-контактний аналоговий роз'єм для підключення монітора до комп'ютера. Це єдиний аналоговий інтерфейс, який застосовується сьогодні. Хоч VGA і не назвеш прогресивним, його все ще можна зустріти в багатьох електронних пристроях (рис 2.45). Головним недоліком даного роз'єму є подвійне перетворення сигналу в аналоговий формат і назад, під час підключення цифрових пристроїв відображення. Це спричиняє погіршення якості зображення. Офіційно, технологія VGA здатна передавати відеосигнал з роздільною здатністю 1280x1024 пікселів, не більше. На практиці, роздільна здатність може досягати Full HD (1920×1080), а в деяких випадках навіть 2048×1536. Але чим вищим буде розширення переданого сигналу, тим більше шансів получити несподівані дефекти зображення.

Рисунок 2.45 – Роз'єм VGA (D-Sub)

**DVI (Digital Visual Interface)** – перший цифровий відео-інтерфейс, призначений для якісної передачі зображення на цифрові пристрої відображення, такі як рідкокристалічні монітори, телевізори та проектори. В сучасних реаліях існують два типи DVI роз'ємів – це DVI-D та DVI-I, які є надзвичайно

популярними. У роз'ємі DVI-D буква «D» означає слово Digital, що в перекладі – цифровий. Це означає, що в даному варіанті немає аналогового каналу та передача зображення відбувається лише в цифровому форматі. Тобто користувач не зможе напряму підключити аналоговий монітор. Цю проблему можна легко вирішити використовуючи відповідний перехідник (DVI/ VGA). В свою чергу, DVI-I – це розширений варіант інтерфейсу DVI-D та містить 2 типи передачі сигналу – цифровий та аналоговий. Даний вихід досить часто використовують для цифрових дисплеїв і відеокарт (рис 2.46).

Рисунок 2.46 – Роз'єм DVI

**HDMI (High Definition Multimedia Interface)** – це цифровий інтерфейс для передачі відео та аудіо даних, створений як альтернатива DVI інтерфейсу. Даний роз'єм присутній практично в усіх сучасних телевізорах, моніторах, ігрових приставках, ноутбуках, проекторах та інших мультимедійних пристроях. Основна відмінність між HDMI та DVI полягає в менших розмірах HDMI та підтримці передачі багатоканальних цифрових аудіо-сигналів. HDMI відрізняється більшою пропускною здатністю ніж його попередники. Кабель може передавати відео у форматі 1080p, 1440p, Full HD 3D, 4K та 8K (рис 2.47).

Рисунок 2.47 – Роз'єм HDMI

**DisplayPort** – відносно новий тип цифрового інтерфейсу для передачі аудіо та відеосигналу до пристрою відображення (монітору комп'ютера). DisplayPort було розроблено як альтернативу для VGA та DVI інтерфейсів. DisplayPort є найпрогресивнішим інтерфейсом на сьогоднішній день. Варто зазначити, що всі кабелі DisplayPort сумісні з будь-якими пристроями, в яких є аналогічний роз'єм. Сумісність не залежить від версії інтерфейсу або рівня сертифікації. Всі доступні функції будуть працювати через будь-який кабель DP. Сьогодні DisplayPort найчастіше використовується для підключення монітору до комп'ютера. З його допомогою можна підключити відразу кілька моніторів до одного комп'ютера. Висока пропускна здатність дозволяє зробити це без будь-яких серйозних

обмежень. Зображення на усіх дисплеях буде відображатися в максимальному розширенні 2К, 4К та 8К (якщо джерело може забезпечити відповідний формат). Крім того, даний інтерфейс може застосовуватися при підключення різної відеотехніки або акустичних систем (рис 2.48) [21].

Рисунок 2.48 – Роз’єм DisplayPort

7. Система охолодження – елемент, від якого багато в чому залежить комфорт використання графічного прискорювача. При виборі краще віддати перевагу виробам, виконаним із застосуванням вакуумних термотрубок (їх видно при візуальному огляді). Такі системи насправді виявляються більш ефективними і створюють набагато менше шуму. Крім того, ефективне охолодження надає можливість краще «розігнати» відеокарту, домогтись більш високих показників її продуктивності. Високоефективну систему охолодження для графічної плати можна придбати окремо, замінивши штатну. Але коштує така система, як правило, не дешево. Тому вигідніше купувати відеокарти з ефективною штатною системою охолодження.

Одним з ефективних способів підвищення продуктивності відеопідсистеми комп’ютера є одночасне використання в одній машині потужностей відразу декількох відеокарт. Для цього потрібна материнська плата з підтримкою такої можливості (з декількома роз’ємами PCI-Express), відеокарти з реалізацією відповідних технологій, високопродуктивний центральний процесор і досить потужний блок живлення (не менше 700-800 Вт).

Технологія одночасного використання декількох графічних плат від NVidia називається SLI (Scalable Link Interface). Аналогічна технологія ATI має назву CrossFireX. При побудові систем на базі цих технологій можливі варіанти з’єднання відеокарт як через спеціальний гнучкий місток, так і на рівні драйвера (без використання гнучкого містка для їх фізичного з’єднання). В останньому випадку продуктивність буде нижче на 10-15%, обмін даними між картками здійснюється через материнську плату [12].

## 2.10 Блок живлення

Блок живлення комп’ютера – джерело електроживлення, призначене для постачання вузлів комп’ютера електричною енергією постійного струму, шляхом перетворення мережевої напруги 220 В до необхідних значень.

В деякій мірі блок живлення також виконує функції стабілізації і захисту від незначних перешкод живлячої напруги та, будучи забезпечений вентилятором, бере участь в охолодженні компонентів персонального комп'ютера.

Комп'ютерний блок живлення для комп'ютера стандарту PC, персонального або ігрового, згідно специфікації ATX, повинен забезпечувати вихідні напруги ± 5, ± 12, +3,3 Вольт, а також +5 Вольт чергового режиму.

Основними силовими лініями є напруги +3,3 В, +5 і +12 В. Причому, чим вище напруга, тим більша потужність передається по даним лініям. Негативні напруги живлення (–5 В і –12 В) допускають невеликі струми і в сучасних материнських платах в даний час практично не використовуються.

Напруга –5 В використовувалася тільки інтерфейсом ISA і через фактичну відсутність цього інтерфейсу на сучасних материнських платах провід –5 В у нових блоках живлення відсутній.

Напруга –12 В необхідна лише для повної реалізації стандарту послідовного інтерфейсу RS-232, тому також часто відсутня.

Напруги ± 5, ± 12, +3,3, +5 В чергового режиму використовуються материнською платою. Для жорстких дисків, оптичних приводів, вентиляторів використовуються тільки напруги +5В і +12 В.

Будова блока живлення наведена на рис 2.49.


Рисунок 2.49 - Імпульсний блок живлення комп'ютера (ATX): A - вхідний діодний випрямляч, нижче видно вхідний дросельний фільтр; B - конденсатори вхідного фільтра, правіше – радіатор високовольтних транзисторів; C – імпульсний трансформатор, правіше видно радіатор низьковольтних діодних випрямлячів; D - дросель групової стабілізації; E - конденсатори вихідного фільтра

**Вхідний фільтр** (дросель і конденсатори) – запобігає поширенню імпульсних перешкод в живильній мережі, зменшує стрибки струму заряду електролітичних конденсаторів при включенні комп'ютера в мережу, який може призвести до пошкодження вхідного випрямного моста.

**Вхідний випрямний міст** перетворює змінну напругу в постійну пульсуючу.

Конденсаторний фільтр згладжує пульсації випрямленої напруги.

Окремий малопотужний блок живлення видає +5 В чергового режиму і +12 В для живлення мікросхеми перетворювача. Зазвичай дане джерело живлення виконане у вигляді перетворювача на дискретних елементах або на типових моделях на мікросхемі TOPSwitch.

Перетворювач півмостовий виконаний на двох біполярних транзисторах. Схема управління перетворювача захисту комп'ютера від перевищення чи зниження живлячих напруг, зазвичай виконана на спеціалізованій мікросхемі TL494, UC3844, KA5800 чи ін.

**Імпульсний високочастотний трансформатор** служить для формування необхідних номіналів напруги, а також для гальванічної розв'язки ланцюгів (вхідних від вихідних, а також, при необхідності, вихідних один від одного). Ланцюги зворотного зв'язку підтримують стабільну напругу на виході блоку живлення.

**Вихідні випрямлячі.** Позитивні та негативні напруги (5 і 12 В) використовують одній ті ж вихідні обмотки трансформатора з різним напрямком включення діодів випрямляча. Для зниження втрат, при великому споживаному струмі, в якості випрямлячів використовують діоди Шотткі, що володіють малим прямим падінням напруги.

**Дросель вихідний групової стабілізації.** Дросель згладжує імпульси, накопичуючи енергію між імпульсами з вихідних випрямлячів. Друга його функція – перерозподіл енергії між ланцюгами вихідних напруг. Так якщо по якомусь каналу збільшиться споживаний струм, що знизить напругу в цьому ланцюзі, дросель групової стабілізації як трансформатор знизить напругу по інших ланцюгах. Ланцюг зворотного зв'язку виявить зниження вихідних ланцюгів, збільшить загальну подачу енергії, і відновить необхідні значення напруг.

**Вихідні фільтруючі конденсатори.** Вихідні конденсатори, разом з дроселем групової стабілізації інтегрують імпульси, тим самим одержуючи необхідні значення напруг, які значно нижче напруг з виходу трансформатора на одну лінію або на кілька ліній (зазвичай +5 і +3,3 В) навантажувальних резисторів 10-25 Ом, для забезпечення безпечної роботи на холостому ходу.

Блок живлення повинен мати достатню потужність, щоб задовольнити «апетити» всіх пристроїв, що входять до складу системи. Для роботи середньостатистичного комп'ютера досить блоку живлення потужністю 500-650 Вт, але завжди краще купувати блок живлення «із запасом». Запас потрібен на випадок подальшого апгрейду (апаратного вдосконалення) комп'ютера шляхом встановлення в нього нових більш потужних або додаткових пристроїв, наприклад, декількох відеокарт, додаткового жорсткого диску. Вже існують блоки живлення потужністю більше 2500 Вт [12].

## 2.11 Зовнішні носії інформації

Зовнішні носії інформації – це пристрої для тривалого зберігання даних (програм, результатів розрахунків, тощо). Носії зовнішньої пам'яті, забезпечують транспортування даних у випадках, коли комп'ютери не об'єднують у мережі (локальні чи глобальні).

На сьогодні з великої кількості різноманітних типів зовнішніх носіїв використовуються оптичні та флеш накопичувачі.

**Оптичний привод** або оптичний накопичувач – електричний пристрій для зчитування і (залежно від конструкції) запису інформації з оптичних носіїв (наприклад, CD-ROM або DVD-ROM).

Будова оптичного приводу (рис 2.50).

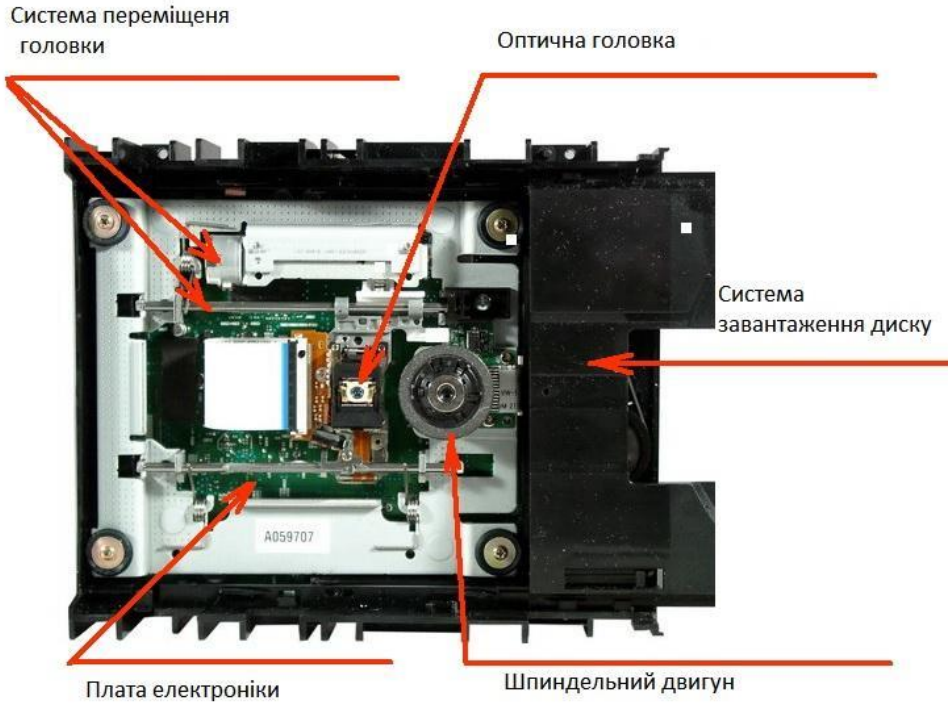



Рисунок 2.50 – Будова оптичного привода

Оптичні приводи розрізняються за підтримуваними форматами лазерних дисків, а також, можливістю запису на оптичний диск.

Так, CD-дисководи підтримують лише формати CD, DVD-дисководи підтримують CD і DVD, а BD-дисководи підтримують формати CD, DVD та BD. З іншого боку зчитувальні приводи (ROM) дозволяють лише зчитувати інформацію, записувальні приводи (recordable) дозволяють зчитувати та записувати відповідні формати дисків, а перезаписувальні (rewritable) – зчитувати, записувати та перезаписувати.

Приводи розрізняють за швидкістю зчитування та запису даних. При цьому за точку відліку (1x) береться швидкість зчитування звукового компакт-диску, що дорівнює 150 кбайт/с. Таким чином 2x – це 300 кБ/с, 4x – 600 кБ/с і т. д. Сучасні пристрої дозволяють зчитувати і записувати CD на швидкостях до 52x,

одношарові DVD до 16х, швидкість перезапису дещо менша (до 32х для CD-RW та 4х для DVD-RW).

Нижче наведено таблицю 2.4 сумісності різних типів оптичних приводів та оптичних дисків [22].

Таблиця 2.4 - Сумісність різних типів оптичних приводів та оптичних дисків

| | CD-ROM | CD-R | CD-RW | DVD-ROM | DVD-R | DVD+R | DVD-RW | DVD+RW | DVD+R DL | BD-ROM | BD-R | BD-RE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Audio CD player | Читає | Читає | | | | | | | | | | |
| CD-ROM drive | Читає | Читає | Читає | | | | | | | | | |
| CD-R recorder | Читає | Пише | Читає | | | | | | | | | |
| CD-RW recorder | Читає | Пише | Пише | | | | | | | | | |
| DVD-ROM drive | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | | | |
| DVD-R recorder | Читає | Пише | Пише | Читає | Пише | Читає | Читає | Читає | Читає | | | |
| DVD-RW recorder | Читає | Пише | Пише | Читає | Пише | Читає | Пише | Читає | Читає | | | |
| DVD+R recorder | Читає | Пише | Пише | Читає | Читає | Пише | Читає | Читає | Читає | | | |
| DVD+RW recorder | Читає | Пише | Пише | Читає | Читає | Пише | Читає | Пише | Читає | | | |
| DVD±RW recorder | Читає | Пише | Пише | Читає | Пише | Пише | Пише | Пише | Читає | | | |
| DVD±RW/DVD+R DL recorder | Читає | Пише | Пише | Читає | Пише | Пише | Пише | Пише | Пише | | | |
| BD-ROM | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає | Читає |
| BD-R recorder | Читає | Пише | Пише | Читає | Пише | Пише | Пише | Пише | Пише | Читає | Пише | Читає |
| BD-RE recorder | Читає | Пише | Пише | Читає | Пише | Пише | Пише | Пише | Пише | Читає | Пише | Пише |

**USB-флеш-накопичувач** (USB Flash drive) – носій інформації, що використовує флеш-пам’ять для збереження даних та підключається до комп’ютера чи іншого пристрою через USB-порт.

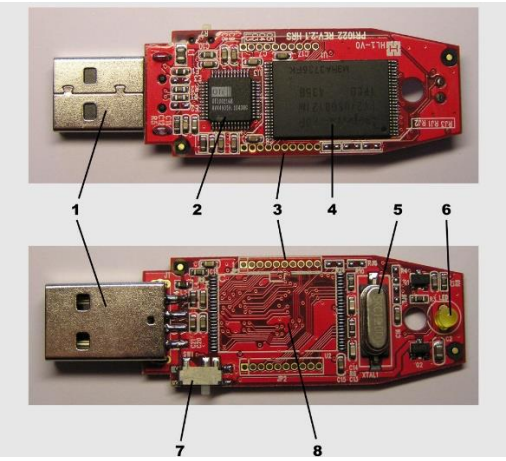



Рисунок 2.51 – Будова USB-флеш-накопичувача. 1 – USB-роз’єм; 2 – мікроконтролер; 3 – контрольні точки; 4 – мікросхема флеш-пам’яті; 5 – кварцовий резонатор; 6 – світлодіод; 7 – перемикач «захист від запису»; 8 – місце для додаткової мікросхеми пам’яті

USB-флеш-накопичувачі зазвичай підтримують перезаписування. Розмір – близько 1 см, вага - менше, ніж 20г. Надзвичайну популярність здобули у 2000-

них у зв’язку з тим, що вони дуже компактні, легкі і мають великий об’єм пам’яті (від 32 Мб до 1Тб). Основне призначення – зберігання й перенесення файлів та обмін ними, резервне копіювання, завантаження операційних систем тощо (рис 2.51) [23].

Сучасні USB-флеш-накопичувачі оснащуються інтерфейсами стандартів USB2, USB3.x, USB Type C, MicroUSB.

**Карта пам’яті або флеш-карта** – компактний електронний носій інформації, що використовується для зберігання цифрової інформації. Сучасні карти пам’яті виготовляються на основі флеш-пам’яті, хоча принципово можуть використовуватися й інші технології. Карти пам’яті широко використовуються в електронних пристроях, включаючи цифрові фотоапарати, стільникові телефони, ноутбуки, MP3-плеєри та ігрові консолі. Карти пам’яті є компактними, перезаписуваними, і, крім того, вони можуть зберігати дані без споживання енергії (енергонезалежність).

Розрізняють карти з незахищеною, повнодоступною пам’яттю, для яких відсутні обмеження на читання і запис даних, і картки з захищеною пам’яттю, що використовують спеціальний механізм дозволів на читання/запис і видалення інформації. Зазвичай картки з захищеною пам’яттю містять незмінну область ідентифікаційних даних.

На даний час найбільш поширеними є карти наступних форматів: CompactFlash, Secure Digital card.

CompactFlash – формат флеш-пам’яті, з’явився одним з перших. Формат розроблений компанією SanDisk в 1994 році. Максимальна місткість накопичувачів з інтерфейсом CompactFlash досягла 32 Гбайт, у 2008 році - 100 Гбайт. Швидкість читання/запису – 160/120 Мб/с. Розміри карт CompactFlash: 43×36×3.3 мм (Type I), 43×36×5 мм (Type II). Використовується в цифрових камерах та інших пристроях (рис 2.52).


Рисунок 2.52 – Карта CompactFlash

Secure Digital Memory Card (SD) – портативна флеш-карта пам’яті, використовується в цифрових фотоапаратах, мобільних телефонах, плеєрах тощо. Розроблена в 2001 році фірмою «SanDisk» на основі MultiMediaCard (MMC). Оригінальний розмір – 24x32x2,1 мм, пізніше з’явилися численні модифікації в бік мініатюризації.

Карта забезпечена власним контролером і спеціальною областю, здатною, на відміну від MMC, записувати інформацію так, щоб було заборонено

незаконне читання інформації відповідно до вимог «Secure Digital Music Initiative», що було закріплено в назві – «Secure Digital». SD використовує спеціальний протокол запису, який недоступний звичайним користувачам.

У більшості випадків SD можна замінити MMC-картою. Заміна у зворотному напрямі зазвичай неможлива, оскільки SD товстіша і може просто не увійти до слота для MMC.

Обсяг пам’яті може бути:

- стандарт SD 1.0 від 8 МБ до 2 ГБ;
- стандарт SD 1.1 можливий розмір до 4 ГБ;
- стандарт SDHC дозволяє місткість до 32 ГБ;
- стандарт SDXC дозволяє місткість до 2 ТБ;
- стандарт SDUC дозволяє місткість до 128 ТБ.

Швидкість обміну SD карт, як і у випадку з CD-ROM задається числом-множником. 1X = 150 Кб/с. Прості карти мають швидкість 6x (900 Кб/с), найновіші – 150X (22500 Кб/с).

Для мініатюрних приладів розроблені mini SD розміром 20x21,5x1,4 мм і найменша зі всіх карт – MicroSD (раніше відома як TransFlash) розміром 11x15x1 мм. Карти MiniSD і MicroSD мають адаптери, за допомогою яких їх можна вставляти в будь-який слот для звичайної SD-карти (рис. 2.53).


Рисунок 2.53 – Карта MicroSD та адаптер SD

## 2.12 Монітори

Монітор (дисплей) комп’ютера – це пристрій, який призначений для візуального відображення текстової і графічної інформації. Його можна сміливо назвати однією з найважливіших частин персонального комп’ютера. Від розміру і якості зображення монітору залежить, наскільки буде комфортна і продуктивна робота за персональним комп’ютером, оскільки користувач постійно контактує з екраном дисплея під час роботи.

В залежності від способу утворення зображення на екрані, монітори діляться на кілька типів:

- на основі електронно-променевої трубки (CRT);
- рідкокристалічні (LCD);
- плазмові;
- проекційні;

- віртуальні ретинальні монітори – технологія пристроїв виведення, що формують зображення безпосередньо на сітківці ока [10].

На даний час найпоширенішими є монітори на базі технології LCD. Рідкокристалічний дисплей (англ. liquid crystal display (LCD)) – електронний пристрій візуального відображення інформації (дисплей), принцип дії якого ґрунтується на явищі електричного переходу Фредерікса в рідких кристалах. Дисплей складається з довільної кількості кольорових або монохромних точок (пікселів) і джерела світла або відбивача (рефлектора).

Кожна з кольорових точок рідкокристалічного дисплея складається з кількох комірок (як правило, з трьох), попереду яких встановлюються світлові фільтри (найчастіше червоний, синій і зелений). Тобто колір певної точки і її яскравість визначається інтенсивностями світіння комірок, з яких вона складається.

Керування кожною рідкокристалічною коміркою здійснюється з допомогою напруги, яку подає на комірку один з транзисторів тонкої підкладки (TFT - абревіатура англійського виразу «Thin Film Transistors»).

Рідкокристалічні дисплеї мають низьке енергоспоживання, тому вони знайшли широке застосування, як в кишенькових пристроях (годинниках, мобільних телефонах, кишенькових комп'ютера), так і в комп'ютерних моніторах, телевізорах тощо.

Екран LCD є масивом маленьких сегментів (пікселів), котрими можна маніпулювати для відображення інформації. LCD має кілька шарів де ключову роль грають дві панелі, зроблені з вільного від натрію і дуже чистого скляного матеріалу, який називають субстратом або підкладкою. Проміжок між шарами заповнений тонким шаром рідкого кристалу. На панелях є борозенки, що надають їм спеціальної орієнтації. Борозенки розташовані паралельно між собою в межах кожної панелі, але борозенки однієї панелі перпендикулярні до борозенок іншої. Поздовжні борозенки утворюються внаслідок нанесення на скляну поверхню тонких плівок прозорого пластику, що потім спеціальним чином обробляється (рис. 2.54).

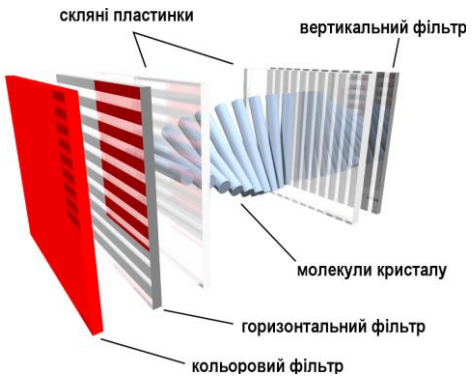



Рисунок 2.54 – Будова LCD екрану

Борозенки орієнтують молекули рідкого кристалу однаково у всіх комірках. Молекули одного з типів рідких кристалів (нематиків) при відсутності напруги повертають вектори електричного (і магнітного) полів світлової хвилі

на деякий кут у площині, перпендикулярній до напрямку поширення світлового променю. Нанесення борозенок на поверхню скла дозволяє забезпечити однаковий кут повороту площини поляризації для всіх комірок. Проміжок між панелями дуже тонкий.

Технічні характеристики:

- тип матриці - технологія виготовлення дисплею;
- роздільна здатність – кількість пікселів у кожному з вимірів, що може бути відображена;
- розмір пікселя – відстань між центрами сусідніх пікселів;
- яскравість – світлова характеристика тіл, які є джерелами світла;
- контрастність – міра виявлення об’єкта на фоні;
- час відгуку – мінімальний час, необхідний пікселю для зміни своєї яскравості;
- кут огляду – кут відносно перпендикуляра до центру матриці, при спостеріганні котрого контрастність зображення у центрі матриці падає до 10:1 [24].

**Технології LCD**. **TFT** – скорочення від Thin Film Transistor. Технологія екранів з активною матрицею. Це звичайний ЖК-екран, але на тонкоплівкових транзисторах. Велика частина моніторів у продажу – це LCD TFT.

**IPS** – рідкокристалічна матриця. Була створена для ліквідації недоліків TN матриці. Технологія збільшила огляд до 178° по вертикалі і горизонталі, її характеризує високий рівень контрастності і хороша передача відтінків. Така матриця дозволяє створити яскраву і чітку картинку. Оптимально підходить для екранів, які використовуються для роботи в інтернеті, перегляду кінострічок, обробки фото.

**TH** – одна з найпростіших технологій матриці. TN плюс film означає додатковий шар, який використовується для забезпечення огляду на 90-170 градусів по горизонталі і 65-160 по вертикалі. Слово film часто упускають в назві, називаючи просто – монітори Т. Вони найбільш бюджетні з усіх описаних вище. Через те, що у таких екранів не ідеальне зображення при перегляді під кутом і передача кольору поступається моніторам на IPS або MVA.

**AMOLED** (англ. Active Matrix Organic Light-Emitting Diode, AMOLED) – технологія створення дисплеїв для мобільних пристроїв, комп’ютерних моніторів і телевізорів. Технологія передбачає використання органічних світлодіодів як світловипромінюючих елементів і активної матриці з тонкоплівкових транзисторів (TFT) для управління світлодіодами.

**MVA** (Multi-domain Vertical Alignment) або **VA** (Vertical Alignment) – найбільш активно розвивається технологія РК-матриць. Виділяється лише широкими кутами і глибоким чорним кольором. Технологія тепер не поступається, а часом навіть перевершує по колірному охопленню IPS. Відгук повільніший, ніж у TN, але швидший, ніж у IPS (3-4 мс). Саме VA-матриці найчастіше використовуються в вигнутих моніторах.

**ADS** (Advanced Dimension Switch) – новий і тому поки що рідкісний конкурент IPS. При таких же широких кутах огляду, ADS коштує значно менше, ніж IPS, але і трохи програє йому по колірному охопленню. Виходить якась

проміжна ланка між зовсім вже дешевими і тьмяними TN-моніторами і дорогими IPS.

**PLS** (Plane to Line Switching) – давній і вже поступово зникає з продажу конкурент IPS. Головна перевага PLS полягає в більшій щільності пікселів, завдяки чому менше помітна сітка. Колірне охоплення і кути огляду приблизно на одному рівні з IPS.

В реалізації монітори можуть мати додаткові особливості: сенсорний екран, вгнутість екрану (для великих діагоналей та співвідношень сторін).

## 2.13 Принтери

Комп'ютерний принтер (англ. printer – друкар) – периферійний друкувальний пристрій, що підключається до комп'ютера (чи очікує підключення накопичувача або мережі) і має змогу друкувати текстову та іншу графічну інформацію на папері (тут розглянемо технології персональних пристроїв).

**Класифікація за принципом роботи**

**Лазерний принтер.** Принцип технології полягає у тому, що на поверхні фотобарабану коротроном (скоротроном) заряду, або валом заряду рівномірно розподіляється статичний заряд, після цього світлодіодним лазером (або світлодіодною лінійкою) на фотобарабані знімається заряд – тим самим на поверхню барабана поміщається приховане зображення. Далі на фотобарабан наноситься тонер, після цього барабан прокочується папером, і тонер переноситься на папір коротроном перенесення, або валом перенесення. Тонер, залежно від знаку його заряду, може притягуватися до поверхні, що зберегла приховане зображення або фону. Після цього папір проходить через блок термозакріплення для фіксації тонера, а фотобарабан очищається від залишків тонера і розряджається у вузлі очищення (рис. 2.55).

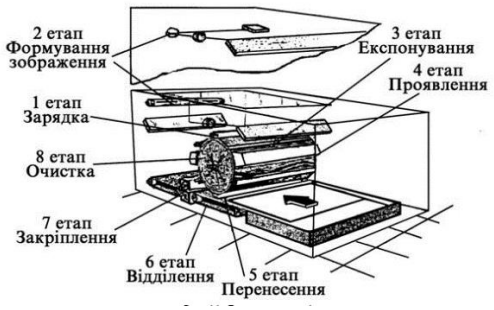



Рисунок 2.55 – Принцип роботи лазерного принтеру

**Струменевий принтер.** Такий принтер формує зображення на носієві з крапок за допомогою матриці, що друкує рідкими барвниками. Картриджі з барвниками бувають із вбудованою друкуючою головкою або друкуюча матриця є деталлю принтера, а змінні картриджі містять тільки барвник. Якщо принтер не використовувати протягом тривалого часу (тиждень і більше), зазвичай

відбувається висихання залишків барвника на соплах друкуючої головки. Струменева технологія передбачає декілька варіантів (рис 2.56).

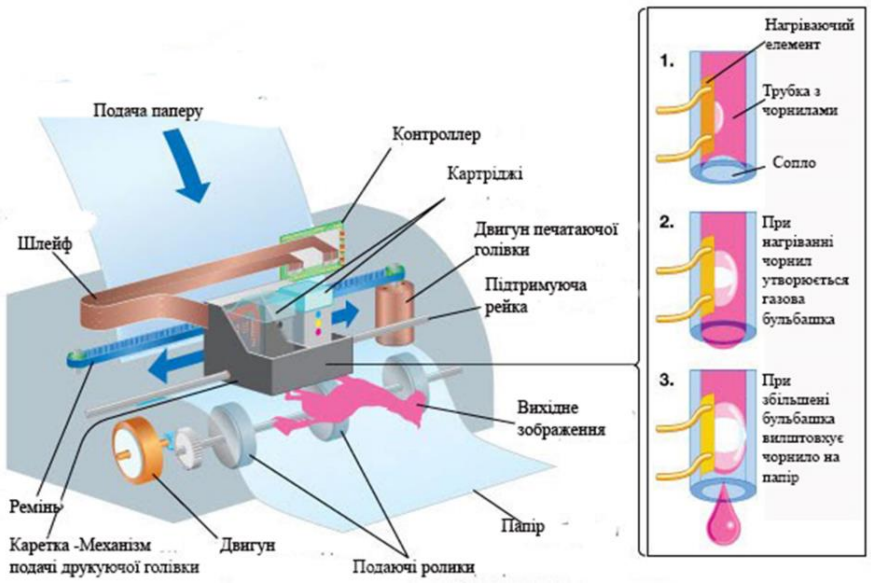



Рисунок 2.56 – Принцип роботи струменевого принтеру

**Сублімаційний принтер**. Термосублімація (сублімація) – це швидкий нагрів фарбника, коли пропускається рідка фаза. З твердого фарбника відразу утворюється пара. Чим менша порція, тим більша фотографічна широта (динамічний діапазон) перенесення кольорів. Пігмент кожного з основних кольорів, а їх може бути три або чотири, знаходиться на окремій (або на загальній багатошаровій) тонкій лавсановій стрічці (термосублімаційні принтери фірми Mitsubishi Electric). Друк остаточного кольору відбувається в декілька проходів: кожна стрічка послідовно протягується під щільно притиснутою термоголовкою, що складається з безлічі термоелементів. Ці останні, нагріваючись, переганяють фарбник. Крапки, завдяки малій відстані між головкою і носієм, стабільно позиціонуються і виходять вельми малого розміру. До серйозних проблем сублімаційного друку можна віднести чутливість вживаного чорнила до ультрафіолету. Якщо зображення не покрити спеціальним шаром, який блокує ультрафіолет, то фарби незабаром вицвітуть. При застосуванні твердих фарбників і додаткового ламінуючого шару з ультрафіолетовим фільтром для оберігання зображення, отримувані відбитки добре переносять вологість, сонячне світло і навіть агресивні середовища, але зростає ціна фотографій. За повноколірність технології сублімації доводиться платити великим часом друку кожної фотографії (рис 2.57).

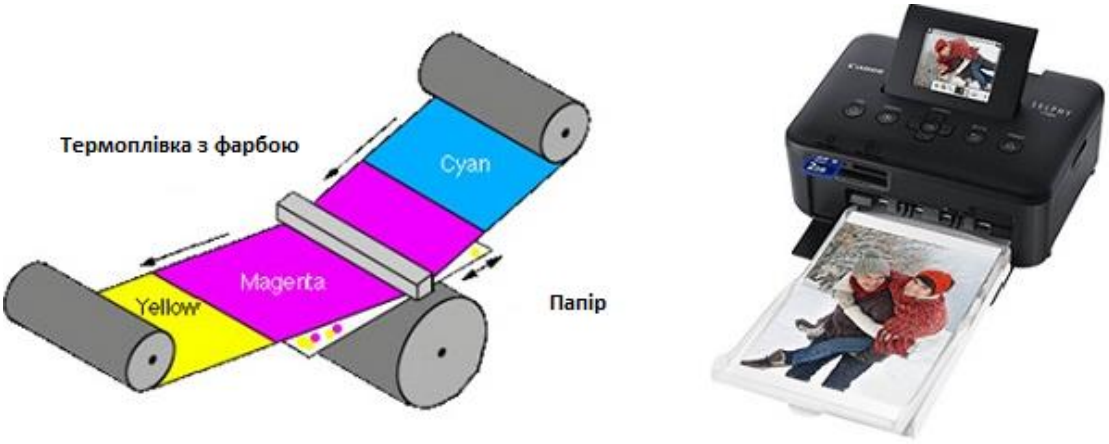



Рисунок 2.57 – Принцип роботи сублімаційного принтера

**Безбарвний термопринтер**. Друк відбувається завдяки нагріву термопаперу. Використовується в касових апаратах. Недолік – термін зберігання копії досить малий і залежать від умов зберігання (рис. 2.58).

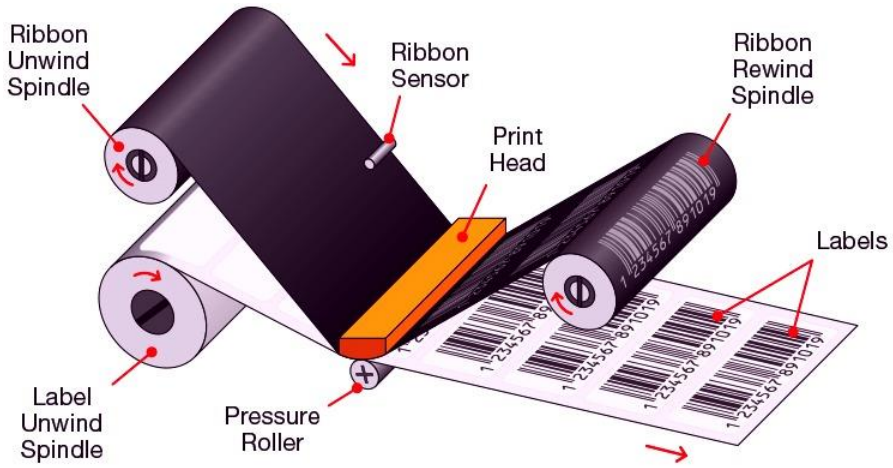



Рисунок 2.58 – Принцип роботи термопринтеру (один з варіантів технології)

**Матричний принтер.** Це найстаріші з нині вживаних типів принтерів. Зображення формується друкуючою головкою, яка складається з набору голок (голкова матриця), що приводяться в дію електромагнітами. Головка пересувається порядково вздовж аркуша, при цьому голки вдаряють по паперу через фарбувальну стрічку, формуючи точкове зображення. Цей тип принтерів називається SIDM (англ. Serial Impact Dot Matrix – послідовні ударно-матричні принтери).

Типовий результат роботи матричного принтеру в режимі draft. Цей малюнок показує фрагмент друку розміром приблизно 4,5×1.5 см

Матричні принтери поширені досі завдяки дешевизні копії (витратним матеріалом, по суті, є тільки фарбувальна стрічка) і можливості роботи з безперервним (рулонним, фальцьованим) і копіювальним папером (рис.2.59).

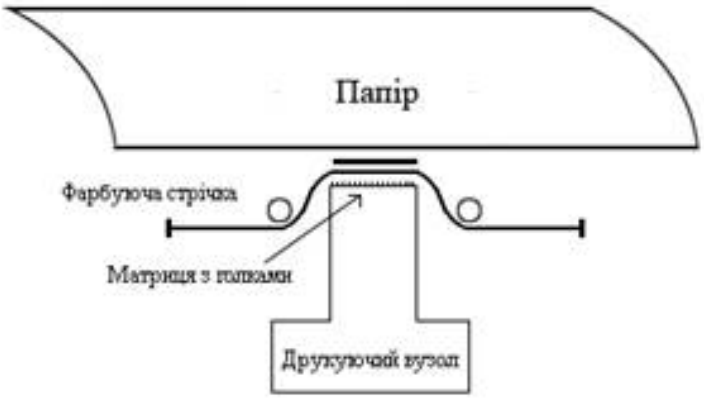



Рисунок 2.59 – Принцип роботи матричного принтеру

**Плоттер**. Широкоформатний принтер. Формат друку від А2 до А0+. Звичайно це струменеві принтери. Іноді зустрічаються інженерні машини на основі лазерного друку.

**Тривимірний принтер**. Принцип роботи 3D принтера заснований на пошаровому відтворенні 3D моделі [25].

На даний час виробники пропонують користувачам багатофункціональні пристрої, які виконують функції сканера, принтера, копіювального апарату. В старших моделях передбачено принт-сервер (BT, Wi-Fi, Ethernet).

## Лабораторна робота № 1
## Дослідження системного інтерфейсу ПК за допомогою системних утиліт

**Мета:** практичне визначення структури та характеристик системного інтерфейсу ПК за допомогою програми AIDA64.

### Порядок виконання роботи

Завантажити програму AIDA64 та за її допомогою скласти у електронному звіті таблицю характеристик компонентів системного інтерфейсу ПК. Останній стовпчик (примітка) у звіті не потрібний – він потрібний тільки для прискорення пошуку необхідного параметра. Зверніть увагу на те, що, залежно від конкретної конфігурації ПК та від версії AIDA64 назви деяких параметрів можуть відрізнятися від вказаних в таблиці 2.5 (або взагалі бути відсутніми).

Таблиця 2.5 – Параметри ПК

| № | Компонент | Поле | Значення | Примітка |
|---|---|---|---|---|
| 1 | Системна плата | Системна плата | | Системна плата |
| 2 | | ID системної плати | | |
| 3 | Властивості шини FSB | Тип шини | | |
| 4 | | Ширина шини | | |
| 5 | | Реальна частота | | |
| 6 | | Ефективна частота | | |
| 7 | Властивості шини пам’яті | Тип шини | | |
| 8 | | Ширина шини | | |
| 9 | | Співвідношення DRAM:FSB | | |
| 10 | | Реальна частота | | |
| 11 | | Ефективна частота | | |
| 12 | | Пропускна здатність | | |
| 13 | Властивості шини чипсету | Тип шини | | |
| 14 | Фізична інформація про системну плату | Число гнізд для ЦП | | |
| 15 | | Роз’єми розширення | | |
| 16 | | Роз’єми ОЗП | | |
| 17 | | Вбудовані пристрої | | |
| 18 | | Форм-фактор | | |
| 19 | | Чипсет системної плати | | |
| 20 | | Північний міст | | |
| 21 | | Підтримувані типи пам’яті | | |
| 22 | | Максимальна кількість пам’яті | | |
| 23 | | Контролер PCI Express | | |
| 24 | | Південний міст | | |
| 25 | | Контролер PCI Express | | |

Скласти в електронному вигляді та подати на перевірку викладачеві звіт

про виконання лабораторної роботи, який містить:

- назву та мету лабораторної роботи;
- заповнену таблицю;
- висновок, що містить стислу характеристику системного інтерфейсу досліджуваного ПК (складові частини, їх особливості та оцінку ступеня збалансованості системного інтерфейсу в цілому).

**Запитання для обговорення отриманих результатів**

1. Пояснити структуру шин ПК, що досліджується.
2. Пояснити склад чипсету та призначення його компонентів.
3. Чим відрізняються поняття реальної та ефективної частоти шини?
4. Яким чином визначається теоретична пропускна здатність шини пам’яті?
5. Як співвідноситься характеристики шини пам’яті та FSB досліджуваного ПК (ширина, реальна частота, ефективна частота, пропускна здатність)? Чи є досліджуваний ПК збалансованим з точки зору пропускної здатності шини пам’яті та FSB?
6. Чим відрізняються (за принципом побудови та функціонування) шини PCI та PCI Express?
7. Чому в сучасних ПК більш розповсюдженими є послідовні шини?

## Лабораторна робота №2
## Дослідження структури та характеристик мікропроцесора

**Мета:** практичне визначення характеристик мікропроцесора ПК за допомогою програми AIDA64.

**Порядок виконання роботи**

Завантажити програму AIDA64 та за її допомогою скласти в електронному звіті таблицю характеристик мікропроцесора. Останній стовпчик (примітка) у звіті не потрібний – він потрібний тільки для прискорення пошуку необхідного параметра. Зверніть увагу на те, що, залежно від конкретної конфігурації ПК та від версії AIDA64 назви деяких параметрів можуть відрізнятися від вказаних в таблиці 2.6 (або взагалі бути відсутніми).

**Запитання**

1. Чи існує єдине рішення задачі пошуку логічної адреси за заданою фізичною адресою? Пояснить свою думку.
2. Який розмір, на ваш погляд, має таблиця дескрипторів у ОЗП ПК на основі МП IA-32? Пояснить свою думку.
3. Що таке параграф? Скільки всього параграфів розташовується у адресному просторі МП x86?
4. Скількома способами (можливими логічними адресами) можна адресувати довільну комірку пам’яті?
5. Яку функцію виконує у МП x86 черга команд? Скільки команд може знаходитися у черзі?

Таблиця 2.6 – Параметри процесора

| № | Характеристика | Значення | Шлях |
|---|---|---|---|
| 1 | Тип ЦП | | ЦП |
| 2 | Псевдонім ЦП | | |
| 3 | Степінг ЦП | | |
| 4 | Набори інструкцій | | |
| 5 | Вихідна частота | | |
| 6 | Мінімальний/ максимальний множник | | |
| 7 | Кеш L1 коду | | |
| 8 | Кеш L1 даних | | |
| 9 | Кеш L2 | | |
| 10 | Кеш L3 | | |
| 11 | Тип корпусу | | |
| 12 | Розміри корпусу | | |
| 13 | Технологічний процес | | |
| 14 | Типова потужність | | |
| 15 | CPU #1 | | |
| 16 | CPU #2 | | |
| 17 | Виробник CPUID | | CPUID |
| 18 | Ім’я ЦП CPUID | | |
| 19 | Версія CPUID | | |
| 20 | Ідентифікатор платформи | | |
| 21 | Версія оновлення мікрокоду | | |
| 22 | Температура Tjmax | | |

## Лабораторна робота № 3-4
## Дослідження підсистеми пам’яті ПК

**Мета:** практичне визначення характеристик підсистеми пам’яті за допомогою програми AIDA64.

### Порядок виконання роботи

1. Завантажити програму AIDA64 (ярлик на робочому столі) та за її допомогою скласти у зошиті таблицю характеристик компонентів підсистеми пам’яті ПК (без стовпчика шлях / примітка).
2. Виконати тест кеш-пам’яті та оперативної пам’яті (меню «Інструменти»).
3. Записати швидкості читання $v_{\text{Read}}$, запису $v_{\text{Write}}$, копіювання інформації $v_{\text{Copy}}$ та значення латентності для основної пам’яті, кеш-пам’яті L1, L2, L3 (при наявності).
4. За результатами тестів (читання, запис, копіювання) знайти середнє значення швидкості передавання інформації $v_{\text{sr}}$ для пам’яті кожного рівня

$$v_{\text{sr}} = \frac{v_{\text{Read}} + v_{\text{Write}} + v_{\text{Copy}}}{3}.$$

5. Визначити, у скільки разів кеш-пам’ять L1, L2, L3 швидше за основну $\left(\frac{v_{\text{sr L1}}}{v_{\text{sr memory}}};\frac{v_{\text{sr L2}}}{v_{\text{sr memory}}};\frac{v_{\text{sr L3}}}{v_{\text{sr memory}}}\right)$ та відповідне співвідношення для рівнів кеш-пам’яті $\left(\frac{v_{\text{sr L1}}}{v_{\text{sr L2}}};\frac{v_{\text{sr L1}}}{v_{\text{sr L3}}};\frac{v_{\text{sr L2}}}{v_{\text{sr L3}}}\right)$.

Таблиця 2.7 – Характеристики компонентів підсистеми пам’яті ПК

| № | Компонент | Поле | Значення | Шлях / Примітка |
|---|---|---|---|---|
| 1 | Кеш- пам’ять | Кеш L1 коду | | *Системна плата/ ЦП/ Властивості ЦП* |
| 2 | | Кеш L1 даних | | |
| 3 | | Кеш L2 | | |
| 4 | Шина пам’яті | Тип шини | | *Системна плата/ Системна плата/ Властивості шини пам’яті* |
| 5 | | Ширина шини | | |
| 6 | | Співвідношення DRAM/FSB | | |
| 7 | | Реальна частота | | |
| 8 | | Ефективна частота | | |
| 9 | | Пропускна здатність | | |
| 10 | Оперативна пам’ять | Ім’я модуля | | *Системна плата/ SPD/ Властивості модуля пам’яті* |
| 11 | | Серійний номер | | |
| 12 | | Дата випуску | | |
| 13 | | Розмір модуля | | |
| 14 | | Тип модуля | | |
| 15 | | Тип пам’яті | | |
| 16 | | Швидкість пам’яті | | |
| 17 | | Ширина модуля | | |
| 18 | | Вольтаж модуля | | |
| 19 | | Метод виявлення помилок | | |
| 20 | | Частота регенерації | | |
| 21 | | Таймінг пам’яті | | *Системна плата/ SPD/ Таймінг пам’яті* Примітка: таймінги для всіх частот, що підтримуються у першому (стандартному) форматі |

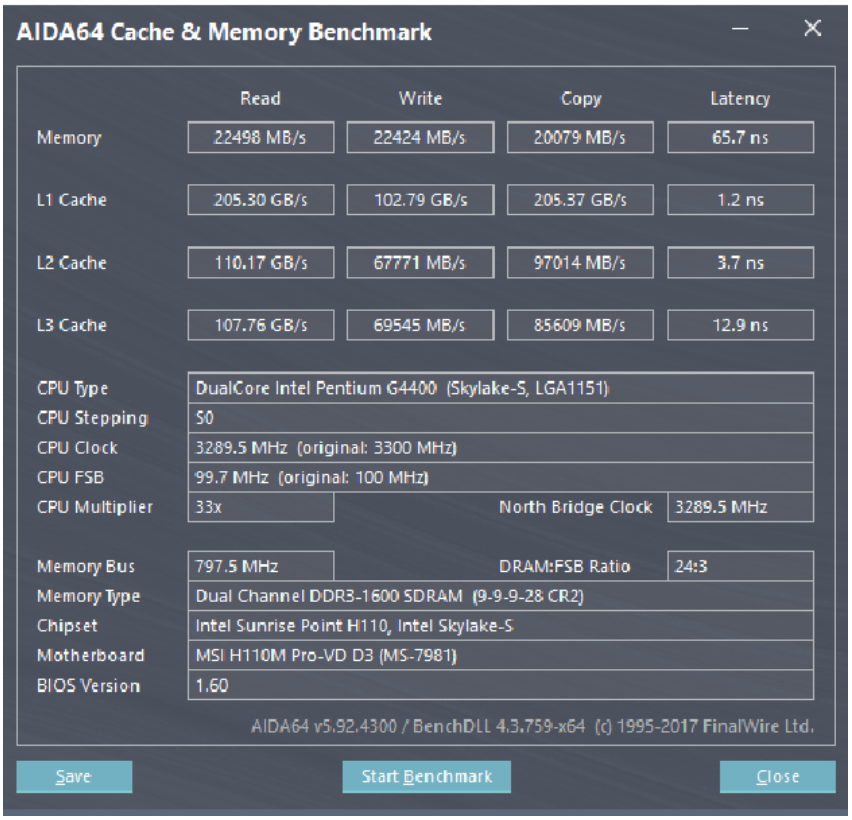



Рисунок 2.60 – Приклад виконання тесту пам’яті

Приклад оформлення результатів розрахунків (для значень, представлених на рис. 2.60).

Таблиця 2.8 – Приклад оформлення результатів розрахунків

| $v_{\text{sr memory}}$ | $v_{\text{sr L1}}$ | $v_{\text{sr L2}}$ | $v_{\text{sr L3}}$ |
|---|---|---|---|
| 22 *Gb/s* | 171 *Gb/s* | 92 *Gb/s* | 88 *Gb/s* |
| | $\frac{v_{\text{sr L1}}}{v_{\text{sr memory}}}$ | $\frac{v_{\text{sr L2}}}{v_{\text{sr memory}}}$ | $\frac{v_{\text{sr L3}}}{v_{\text{sr memory}}}$ |
| | 7,8 | 4,2 | 4 |
| | $\frac{v_{\text{sr L1}}}{v_{\text{sr L2}}}$ | $\frac{v_{\text{sr L1}}}{v_{\text{sr L3}}}$ | $\frac{v_{\text{sr L2}}}{v_{\text{sr L3}}}$ |
| | 1,9 | 2 | 1,05 |

Пояснити отримані результати.

Визначити відношення (у відсотках) реальної пікової пропускної здатності підсистеми основної пам’яті (швидкості читання $v_{\text{Read memory}}$) до теоретичної пропускної здатності шини пам’яті.

Для наведеного прикладу теоретична пропускна здатність складає 25600 Мбайт/с, $v_{\text{Read memory}} = 22\,498$ Мбайт/с, тобто вказане відношення складає приблизно 88%.

**Контрольні запитання та питання для обговорення**

1. Пояснити структуру кеш-пам'яті дослідженої системи.
2. Як визначається теоретична пропускна здатність шини пам'яті?
3. Як співвідносяться середні значення швидкості передавання інформації основної пам'яті та кеш-пам'яті різних рівнів? Які фактори впливають, на ваш погляд, на швидкодію різних рівнів підсистеми пам'яті?
4. Що називають таймінгами оперативної пам'яті, як їх значення впливають на її швидкодію?
5. Що називають латентністю пам'яті? Як значення цього показника впливають на її швидкодію?
6. Оцініть, на скільки зменшилась би реальна пікова пропускна здатність підсистеми основної пам'яті (швидкість читання $v_{\text{Read memory}}$) при функціонуванні оперативної пам'яті в одноканальному режимі?
7. Зробить висновок, чи є необхідність у прискоренні функціонування підсистеми пам'яті даного ПК? Яким чином це може бути зроблено?

## Лабораторна робота №5-6
## Дослідження накопичувача та підсистеми віртуальної пам'яті

**Мета дослідження:** практичне визначення основних характеристик та оцінювання технічного стану накопичувача інформації на жорсткому магнітному диску за допомогою програми AIDA-64.

**Теоретичні відомості (поняття про S.M.A.R.T-контроль)**

Сучасні технології виробництва ЖМД не дозволяють виготовляти їх без дефектів поверхні. Крім того, у процесі експлуатації відбувається старіння магнітного покриття та механічних частин НЖМД. Тому виробники НЖМД запропонували набір технологій, що дозволяють не тільки «приховувати» дефекти поверхні (bad-сектори) та здійснювати постійний моніторинг стану параметрів НЖМД, але й передбачати появу помилок та пошкоджень накопичувачів. Це ускладнило логічну організацію НЖМД. Виникла необхідність використання спеціального транслятора фізичного простору ЖМД у логічний з метою «приховання» від користувача деяких спеціальних областей диска.

Логічний простір – це, фактично, робоча область диска, що доступна користувачу. Типова організація логічного простору НЖМД показана на рис. 2.61.

Крім службової та робочої областей на диску є резервна область, сектори якої призначені для заміни пошкоджених секторів робочої області. При заводському тестуванні НЖМД в його службовій області створюється таблиця дефектів (Primary List або P-List), у яку записуються адреси дефектних секторів робочої області. При форматуванні низького рівня дефектні сектори ігноруються транслятором (не отримують логічних адрес), тому логічний адресний простір секторів робочої області є безперервним, а дефектні області – недоступними. Для забезпечення стандартної інформаційної ємності НЖМД транслятор додає до робочої області частину секторів резервної області. Тому всі нові ЖМД мають

«бездефектну» поверхню і інформаційний обсяг, заявлений виробником.

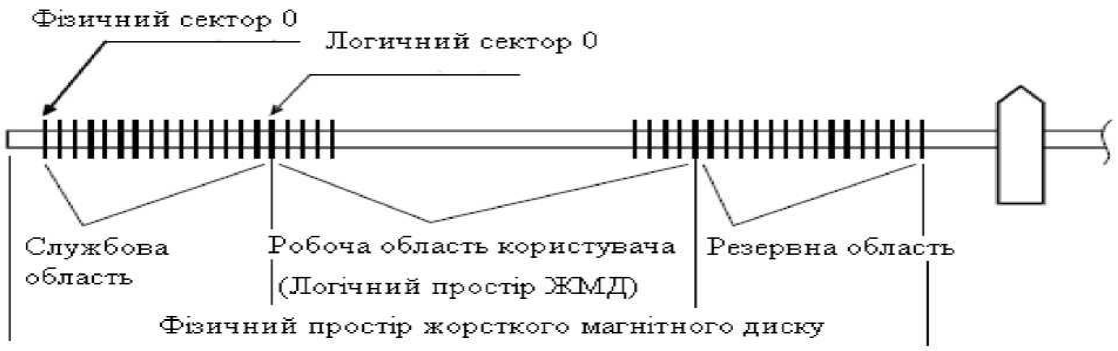



Рисунок 2.61 - Типова організація робочого простору НЖМД.

Якщо дефектні сектори виникають у процесі експлуатації НЖМД, заміна їх виконується автоматично, але такі випадки реєструються системою контролю S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology – виробничий стандарт, створений у 1995 році, який описує методи передбачення помилок НЖМД). Якщо система S.M.A.R.T. активізована, вона стежить за параметрами НЖМД і, на основі тенденції їх змінювання, визначає ймовірність суттєвих збоїв у найближчому майбутньому.

Якщо ця імовірність висока, S.M.A.R.T. генерує попередження, яке вказує користувачеві на необхідність резервного копіювання даних.

Для кожного параметра, який контролюється S.M.A.R.T., виробником встановлюється вихідне значення («ідеальний стан» по даному параметру, найчастіше використовуються значення 255 або 100 «умовних одиниць»). Це значення у процесі функціонування може поступово зменшуватися під впливом різних факторів. Друге значення кожного параметра, що встановлюється виробником і не може бути зміненим – це граничне, «найгірше» значення. Досягнення деяким параметром граничного значення не означає негайної «катастрофи» з втратою даних, але вказує на неприпустиме зниження надійності НЖМД.

Наприклад:

- Raw Read Error Rate – частота появи помилок при читанні з диска;
- Reallocated Sectors Count – кількість «схованих» в процесі експлуатації диска пошкоджених секторів;
- Seek Error Rate – кількість помилок позиціювання БМГ;
- Power-On Hours або Power-On Time Count – загальна кількість часу знаходження у включеному стані.

На рис. 2.62 показаний перелік S.M.A.R.T.-параметрів НЖМД, що знаходиться у повністю справному стані.

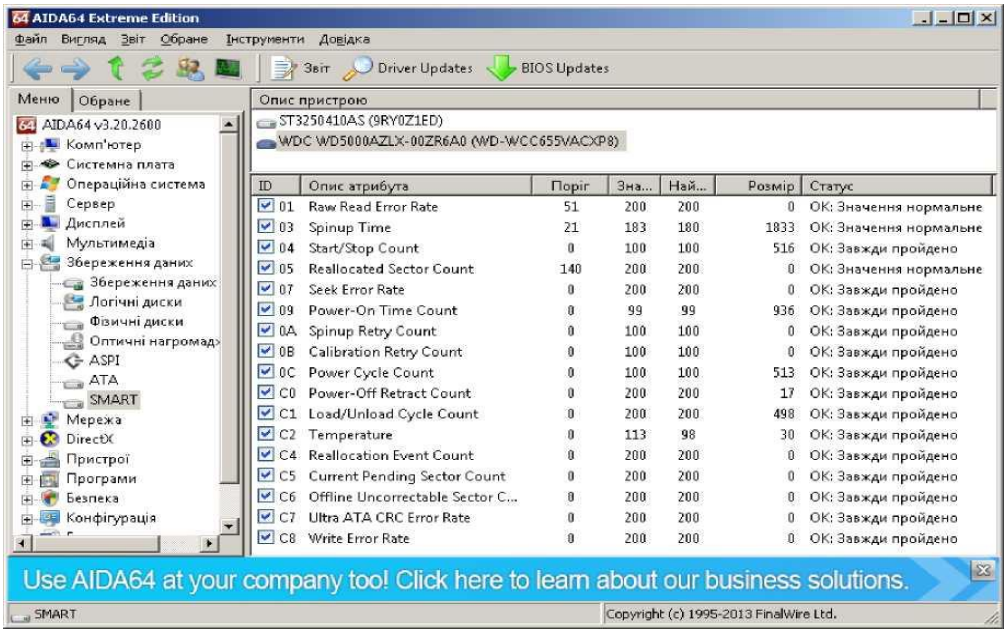



Рисунок 2.62 - S.M.A.R.T.-параметри НЖМД, що знаходиться у повністю справному стані

На рис 2.63 показаний перелік S.M.A.R.T.-параметрів НЖМД після виконання операції *remap* утилітою Victoria. На диску було «сховано» понад 250 пошкоджених секторів, але параметр Reallocated Sectors Count має значення 1. Саме тому, при оцінюванні стану НЖМД, головне значення має реальне напрацювання НЖМД (параметр Power-On Hours або Power-On Time Count). Слід пам’ятати, що цілісність даних гарантується, у більшості випадків при наробітку НЖМД до 5000 годин.

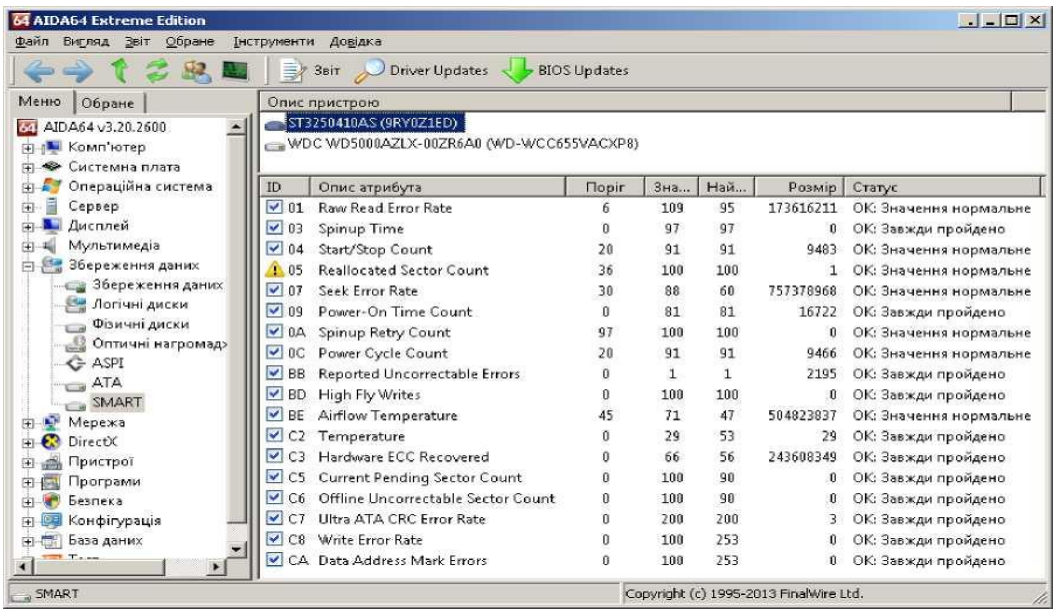



Рисунок 2.63 – Перелік S.M.A.R.T.-параметрів НЖМД після виконання операції remap утилітою Victoria. На диску було «сховано» понад 250 пошкоджених секторів

**Порядок виконання роботи**

1. Завантажити програму AIDA64 та за її допомогою скласти таблицю 2.9 характеристик віртуальної пам’яті (перший набір значень).

Таблиця 2.9 - Характеристики віртуальної пам’яті

| № | Компонент | Поле | Значення | | Шлях / Примітка |
|---|---|---|---|---|---|
| | | | 1 | 2 | |
| 1 | Фізична пам’ять | Усього | | | *Системна плата/ пам’ять* |
| 2 | | Зайнято | | | |
| 3 | | Вільно | | | |
| 4 | | Завантаження | | | |
| 5 | Місце під файл підкачування | Усього | | | |
| 6 | | Зайнято | | | |
| 7 | | Вільно | | | |
| 8 | | Завантаження | | | |
| 9 | Віртуальна пам’ять | Усього | | | |
| 10 | | Зайнято | | | |
| 11 | | Вільно | | | |
| 12 | | Завантаження | | | |
| 13 | Файл підкачування | Файл підкачування | | | |
| 14 | | Поточний розмір | | | |
| 15 | | Поточний/пік використання | | | |
| 16 | | Завантаження | | | |

2. Завантажити 2-3 програми (наприклад, Word, Excel, STDU Viewer). Записати у таблицю другий набір значень. Пояснити, які значення змінилися й чому. Заповнити таблицю 2.10 характеристик накопичувача

Таблиця 2.10 - Характеристики накопичувача

| № | Компонент | Поле | Значення | Шлях / Примітка |
|---|---|---|---|---|
| 1 | Властивості пристрою ATA, фізичні дані | ID моделі | | *Збереження даних / ATA* |
| 2 | | Серійний номер | | |
| 3 | | Тип пристрою | | |
| 4 | | Форматована ємність | | |
| 5 | | Параметри диску | | |
| 6 | | Сектори LBA | | |
| 7 | | Розмір фізичного / логічного сектору | | |
| 8 | | Буфер (обсяг) | | |
| 9 | | Макс. режим PIO | | |
| 10 | | Макс. режим UDMAC | | |
| 11 | | Неформатована ємність | | |
| 12 | | Швидкість обертання | | |
| 13 | | Інтерфейс | | *Збереження даних* |
| 14 | | Сектори LBA | | *ATA* |
| 15 | | Максимальна внутрішня швидкість даних | | |
| 16 | | Неформатована ємність | | |
| 17 | | Швидкість даних буфер-контролер | | |

Порівняйте значення параметрів «форматована ємність» та «неформатована ємність», «диски», «записувані поверхні» та «параметри», розмір фізичного та логічного сектору. Зробить висновки про фізичні та логічні

характеристики НЖМД (SSD) , його швидкодію (швидкість передачі інформації та латентність порівняно з основною пам’яттю, для порівняння використовуйте дані, отримані на попередній лабораторній роботі).

3. Оцінити стан диска за допомогою S.M.A.R.T-тесту (Збереження даних/ S.M.A.R.T). При наявності критичних значень S.M.A.R.T-параметрів НЖМД (SSD) повідомити про це викладача.

Таблиця 2.11 – Значення S.M.A.R.T

| № | Атрибут | Значення | Примітка |
|---|---|---|---|
| 1 | Raw Read Error Rate | | Кількість помилок функціонування різних типів |
| 2 | Seek Error Rate | | |
| 3 | Write Error Rate | | |
| 4 | Hardware ECC Recovered | | |
| 5 | Reported Uncorrectable Errors | | |
| 6 | Ultra DMA CRC Error Count | | |
| 7 | Current Pending Sector Count | | Кількість нестабільних, помилкових та переміщених секторів |
| 8 | Uncorrectable Sector Count | | |
| 9 | Reallocated Sector Count | | |
| 6 | Power On Hours Count | | Напрацювання НЖМД (SSD) (у годинах) та параметри, що характеризують стан його електромеханічної частини |
| 7 | Start/Stop Count | | |
| 8 | Spin up Retry Count | | |
| 9 | Spin up Time | | |

4. Виконати «тест ефективності диску» в режимі «Linear Read» (меню «Інструменти»). Тест виконувати не менше 10 хвилин. Визначити середню швидкість лінійного читання та пояснити характер отриманого графіка.

**Контрольні запитання та питання для обговорення**

1. Стисло поясніть, що таке віртуальна пам’ять, з яких компонентів вона складається?

2. Порівняйте значення параметрів «форматована ємність» та «неформатована ємність», розмір фізичного та логічного сектору. Зробіть висновки про фізичні та логічні характеристики НЖМД, його швидкодію (швидкість передачі інформації та латентність порівняно з основною пам’яттю). Для порівняння використовуйте дані, отримані на попередній лабораторній роботі).

Пояснить основну ідею системи контролю S.M.A.R.T. Які S.M.A.R.T.-параметри, на ваш погляд, є найбільш важливими?

## Лабораторна робота № 7
## Дослідження підсистеми введення-виведення ПК

**Мета:** практичне визначення складу та характеристик підсистеми введення-виведення ПК за допомогою програми AIDA-64.

### Порядок виконання роботи

1 Завантажити програму AIDA-64 та за її допомогою скласти таблицю 2.12 ресурсів компонентів підсистеми введення-виведення ПК (розділ «Пристрої WINDOWS»).

Таблиця 2.12 - Ресурси компонентів підсистеми введення-виведення ПК

| **Пристрій** | **Ресурси пристроїв** | | |
|---|---|---|---|
| | Адреси та розрядність портів (у байтах) | | Номери переривань (IRQ) |
| | Порт | Діапазон пам’яті | |
| *Контролери запам’ятовуючих пристроїв* | | | |
| Serial ATA Controller | | | |
| SAS і RAID-контролери | | | |
| *Системні пристрої* | | | |
| Контролер прямого доступу до пам’яті | | | |
| CMOS та годинник | | | |
| Вбудований динамік | | | |
| Системний таймер та високоточний таймер подій | | | |
| Програмований контролер переривань | | | |
| PCI Express Root Complex | | | |
| Контролер HDA | | | |
| Арифметичний сопроцесор | | | |
| Порт PCI Express Root | | | |
| *Інші пристрої* | | | |
| Відеоадаптер | | | |
| Клавіатура | | | |
| Пристрої безпеки | | | |
| Мережеві пристрої | | | |

### Відповісти на запитання

1. Які пристрої використовують канали DMA? Для чого?

2. Порівняйте ресурси, які використовують контролер Serial ATA, SCSI-контролер, канали IDE. Які висновки, на ваш погляд, можна зробити щодо

пропускної здатності та режимів функціонування різних каналів зв’язку накопичувачів інформації з ядром обчислювальної системи?

3. Які пристрої використовують порти найменшої розрядності (1-2 байти) та найбільшої розрядності (32 або більше)? Чому, на ваш погляд, відрізняються розрядності використовуваних портів?

4. Які пристрої не використовують переривань? Чому, на ваш погляд?

5. Скільки каналів DMA має ПК? Скільки з них використовуються?

6. Пояснити поняття ізольованого введення-виведення та по аналогії з пам’яттю. Навести приклади пристроїв, що використовують дані методи введення-виведення.

# ЧАСТИНА ІІІ
# ПРОГРАМУВАННЯ ЦІЛОЧИСЛОВОГО ПРОЦЕСОРА X86 (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 вісім 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 16384 ($2^{14}$) 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 [26].

## 3.1 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. 3.1) 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.

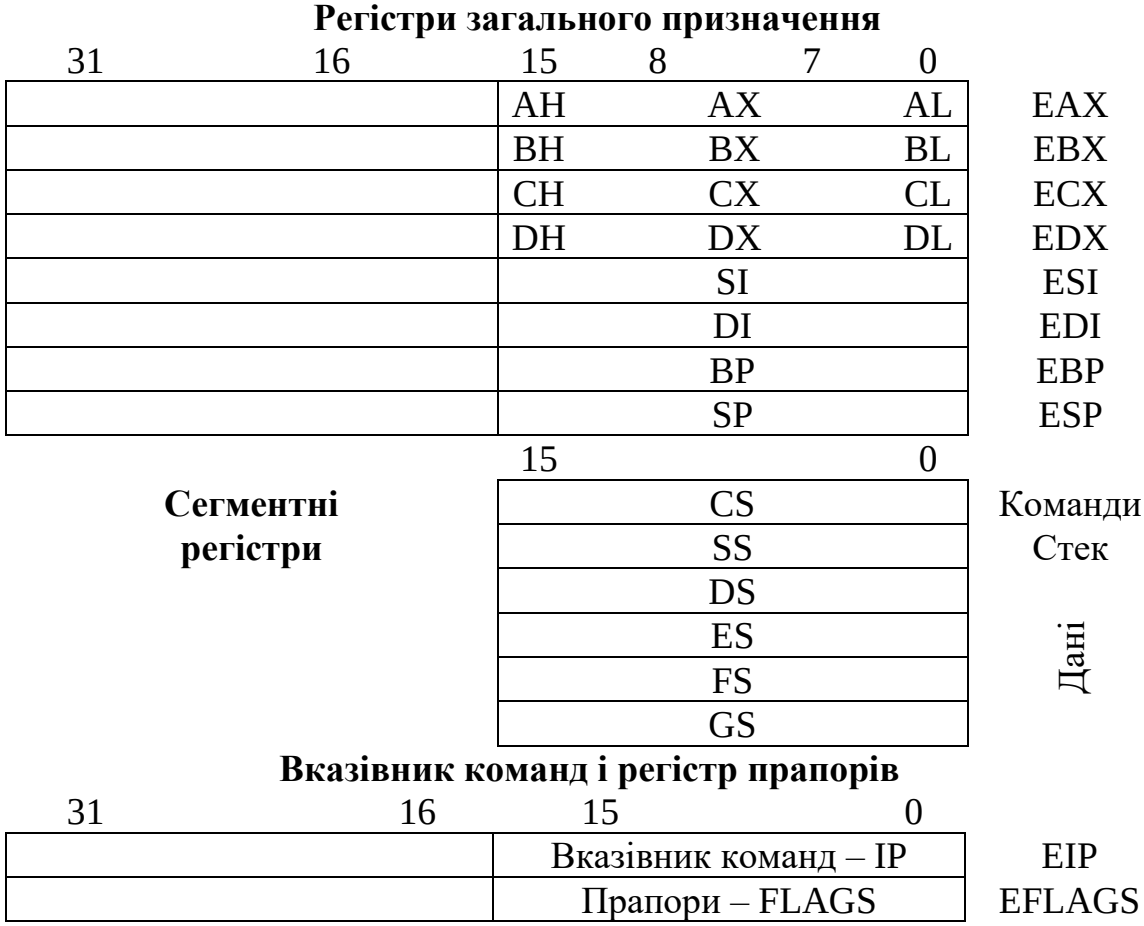



Рисунок 3.1 – Регістри 32-х розрядних мікропроцесорів (386+)

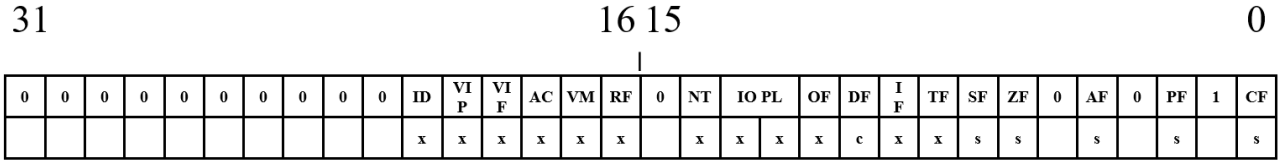
Архітектура мікропроцесора 386+ дозволяє безпосередньо звертатися до 6 сегментів (розміром до 4 Гб кожен) за допомогою спеціальних селекторів, що завантажуються в сегментні регістри програмно. Вміст регістрів загального призначення, селекторів, покажчика команд і регістра прапорів (ознак) залежить від 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. 3.2) відображає стан мікропроцесора. При використанні тільки 16-ти молодших розрядів регістр прапорів сумісний з попередніми моделями мікропроцесорів.




| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ID | VIP | VIF | AC | VM | RF | 0 | NT | IO PL | | 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 3.2 – Регістр прапорів EFLAGS (ознак)

На рис. 3.2 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 Interrupt 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 шість 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. 3.3). Кожен дескрипторний регістр зберігає 32-х бітову базову адресу сегмента, 20-ти бітовий розмір сегмента та інші необхідні його атрибути.

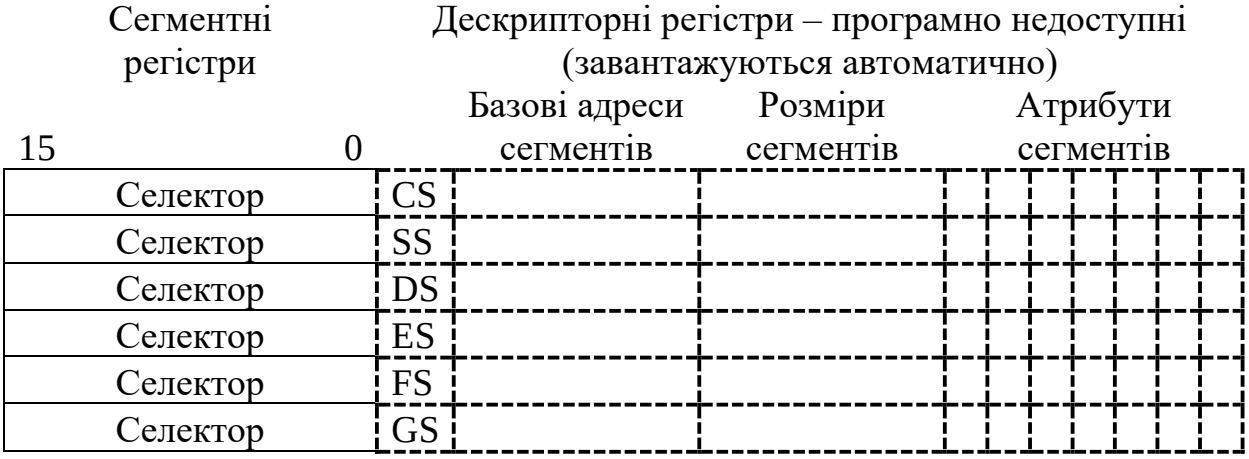



Рисунок 3.3 – Сегментні регістри і відповідні дескрипторні регістри мікропроцесора 386+

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) [26].

## 3.2 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. 3.4). ***Двійкові цілі числа зі знаком*** подаються у додатковому коді. Старший біт є знаковим (рис.3.4): 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. 3.4). У неупакованому форматі байт містить одну десяткову цифру, яка зазвичай зображується в символьному коді ASCII.

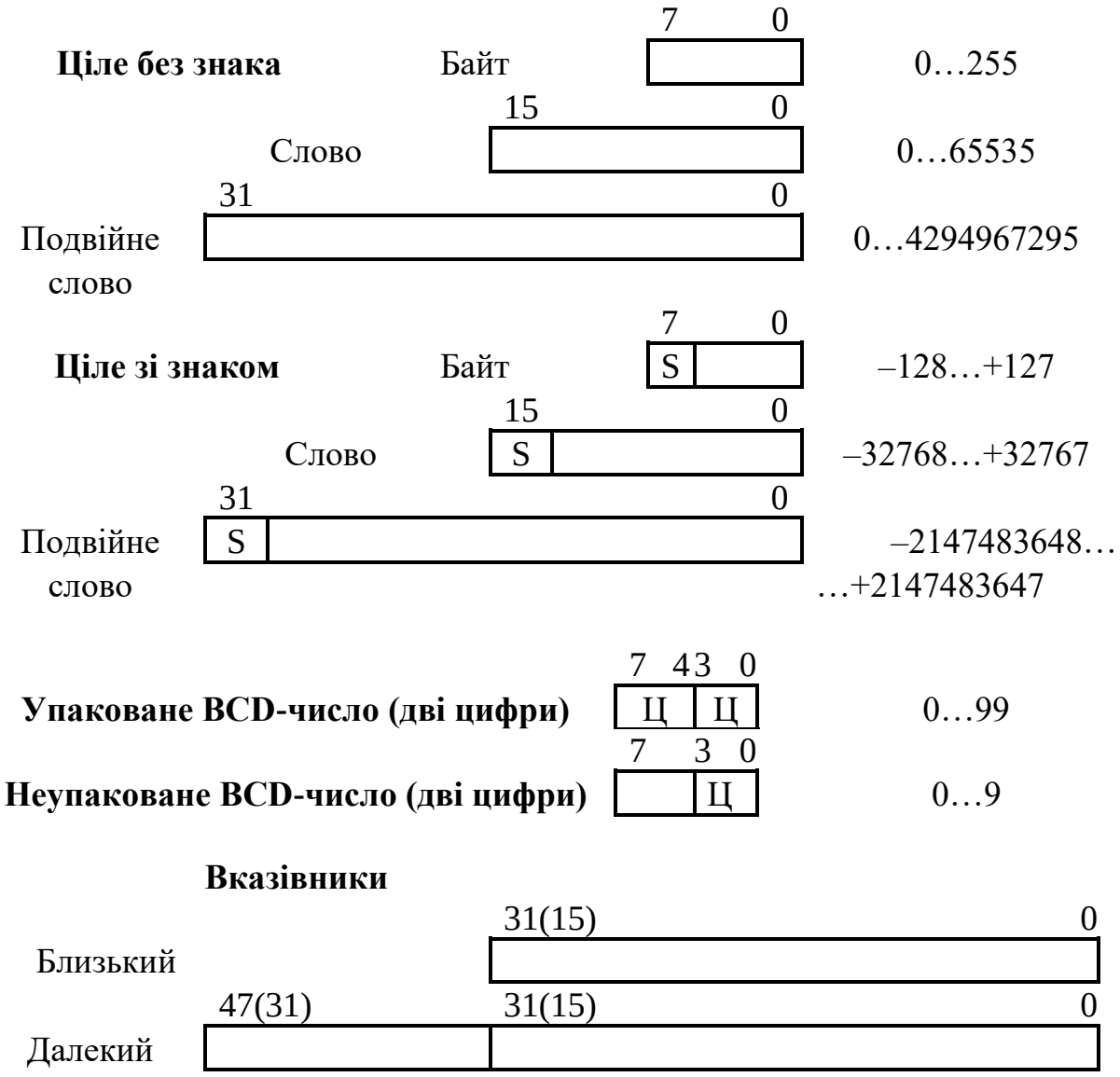



Рисунок 3.4 – Типи даних 32-х розрядних процесорів

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 [26].

## 3.3 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).

**3.3.1 Режими (методи) адресації.** Процесори 386+ забезпечують 13 режимів адресації, що розраховані на ефективне виконання програм, написаних на мовах високого рівня типу: С++, 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 [27].

**3.3.2 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 3.1 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 [26].

Таблиця 3.1 – 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 |

**3.3.3 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. 3.2. 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. 3.2), зокрема **FS** і **GS**.

Таблиця 3.2 – 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 |

## 3.4 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;
- – 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 [27].

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 стеку. 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.

Таблиця 3.3 – 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 зі заповненням старших бітів нулем (386+) |
| MOVZXB reg, r/m | Копіювання слова з розширенням до подвійного слова зі заповненням старших бітів 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 |

Таблиця 3.4 – Команди роботи зі стеком

| | |
|---|---|
| PUSH r/m | Поміщення (подвійного) слова з регістра або пам’яті в стек |
| PUSH immed | 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** 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 вміст регістра (E)SP записаний в стеку.

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 [27].

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. 3.5). Після команди **CMP** зазвичай використовується команда умовного переходу.

Таблиця 3.5 – 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 позикою: 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 | Знакове 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 |

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 3.6 – 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. 3.7), а другий співмножник вказаний в команді. Результат множення вдвічі довший за операнди.

Таблиця 3.7 – 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. 3.8) повинна вдвічі перевищувати розрядність дільника, зазначеного в команді.

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

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.

Таблиця 3.8 – 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 |

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) [27].

Таблиця 3.9 – 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. 3.10) 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).

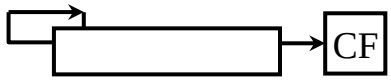
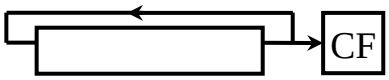
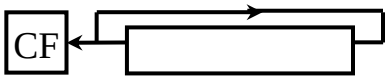
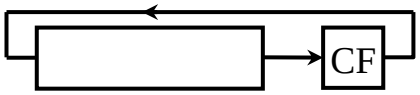
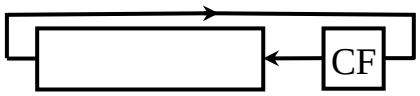
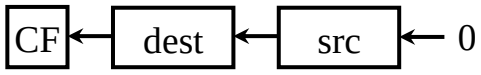
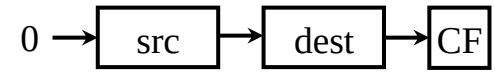
***Команди бітових операцій*** відсутні у мікропроцесорів 86/286.

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.

Таблиця 3.10 – Команди зміщень

| Команда | Мнемоніка | Виконання команди |
|---|---|---|
| Логічне зміщення вліво<br>Арифметичне зміщення вліво | SHL<br>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 |  |

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.

***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. 3.12).

Таблиця 3.11 – Команди бітових операцій (386+)

| | |
|---|---|
| BT r/m, im8<br>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<br>BTC r/m, reg | Тестування (завантаження в CF) і інверсія біта |
| BTR r/m, im8<br>BTR r/m, reg | Тестування (завантаження в CF) і скидання біта |
| BTS r/m, im8<br>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. |

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 3.12 – 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) [27].

Таблиця 3.13 – 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. 3.14) 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 3.14 – 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<br>(LOOPZ) target | Декремент (E)CX і перехід, якщо (E)CX <> 0 & ZF = 1 |
| LOOPNE target<br>(LOOPNZ) target | Декремент (E)CX і 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 «ссс» повністю збігається з умовою переходів (табл. 3.15).

Таблиця 3.15 – 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. 3.16) 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 3.16 – 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 № 8
## Обчислення значення функції

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: $y = \dfrac{15x^2 - 12}{4x + 5} \quad (x = 3)$. 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.

Таблиця 3.17 - Індивідуальні завдання

| Варіант | Функція | Значення $x$ | Варіант | Функція | Значення $x$ |
|---|---|---|---|---|---|
| 1 | $y=5x^2+2x-14$ | 3 | 14 | $y=\frac{2x^2-2x-17}{4+x^2}$ | 6 |
| 2 | $y=\frac{7500}{2x^2+15}$ | 5 | 15 | $y=\frac{4300x}{\frac{1}{3}x^2-15}$ | 9 |
| 3 | $y=\frac{6x^2+12}{5x-8}$ | 7 | 16 | $y=\frac{11x^2-12}{5x^2+8x-2}$ | 4 |
| 4 | $y=\frac{2500x-8}{3x^2+20}$ | 2 | 17 | $y=\frac{370x^2-8}{2x-40}$ | 1 |
| 5 | $y=\frac{8x^2+12x-7}{3x+25}$ | 4 | 18 | $y=\frac{6x^2-10x+7}{3x^2-5}$ | 5 |
| 6 | $y=7x^2+12x-32$ | 8 | 19 | $y=\frac{4x^2-12x-32}{x^2-51}$ | 7 |
| 7 | $y=\frac{5x^2-2x-14}{1-x^2}$ | 11 | 20 | $y=\frac{3x^2+7x-14}{1+x^2}$ | 6 |
| 8 | $y=\frac{3x^2+2x+14}{1+x}$ | 2 | 21 | $y=\frac{4300}{\frac{7}{3}x^2+5}$ | 8 |
| 9 | $y=\frac{2300}{\frac{2}{3}x^2-15}$ | 8 | 22 | $y=\frac{3x-12}{5x^2-8}$ | 12 |
| 10 | $y=\frac{4x^2-12}{5x^2+8}$ | 2 | 23 | $y=\frac{230x+8}{2x^2+20}$ | 4 |
| 11 | $y=\frac{230x^2+8}{2x+20}$ | 5 | 24 | $y=\frac{11x^2-10x+7}{3x^2-5}$ | 2 |
| 12 | $y=\frac{6x^2-10x-7}{3x-5}$ | 4 | 25 | $y=\frac{4300x^2}{\frac{11}{3}x^2-15}$ | 6 |
| 13 | $y=\frac{7x^2+12x-32}{x}$ | 2 | | | |

## Лабораторна робота №9
## Обчислення кількох значень функцій

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: обрахувати 7 значень функції $y=\frac{15x^2-12}{4x+5}$ $(x=3)$. 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                   ; частне - 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.

е. Тест виконання індивідуального завдання (порівняння результату з іншою програмою).

Приклад оформлення звіту наведено у додатку 1.

Таблиця 3.18 - Індивідуальні завдання

| Варіант | Функція | Значення $x$ | Варіант | Функція | Значення $x$ |
|---|---|---|---|---|---|
| 1 | $y=5x^2+2x-14$ | 3 | 14 | $y=\frac{2x^2-2x-17}{4+x^2}$ | 6 |
| 2 | $y=\frac{7500}{2x^2+15}$ | 5 | 15 | $y=\frac{4300x}{\frac{1}{3}x^2-15}$ | 9 |
| 3 | $y=\frac{6x^2+12}{5x-8}$ | 7 | 16 | $y=\frac{11x^2-12}{5x^2+8x-2}$ | 4 |
| 4 | $y=\frac{2500x-8}{3x^2+20}$ | 2 | 17 | $y=\frac{370x^2-8}{2x-40}$ | 1 |
| 5 | $y=\frac{8x^2+12x-7}{3x+25}$ | 4 | 18 | $y=\frac{6x^2-10x+7}{3x^2-5}$ | 5 |
| 6 | $y=7x^2+12x-32$ | 8 | 19 | $y=\frac{4x^2-12x-32}{x^2-51}$ | 7 |
| 7 | $y=\frac{5x^2-2x-14}{1-x^2}$ | 11 | 20 | $y=\frac{3x^2+7x-14}{1+x^2}$ | 6 |
| 8 | $y=\frac{3x^2+2x+14}{1+x}$ | 2 | 21 | $y=\frac{4300}{\frac{7}{3}x^2+5}$ | 8 |
| 9 | $y=\frac{2300}{\frac{2}{3}x^2-15}$ | 8 | 22 | $y=\frac{3x-12}{5x^2-8}$ | 12 |
| 10 | $y=\frac{4x^2-12}{5x^2+8}$ | 2 | 23 | $y=\frac{230x+8}{2x^2+20}$ | 4 |
| 11 | $y=\frac{230x^2+8}{2x+20}$ | 5 | 24 | $y=\frac{11x^2-10x+7}{3x^2-5}$ | 2 |
| 12 | $y=\frac{6x^2-10x-7}{3x-5}$ | 4 | 25 | $y=\frac{4300x^2}{\frac{11}{3}x^2-15}$ | 6 |
| 13 | $y=\frac{7x^2+12x-32}{x}$ | 2 | | | |

Значення $x$ та крок вводяться з клавіатури.

## Лабораторна робота № 10
## Організація умовних переходів

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання: визначити номер ($n$) елемента прогресії: $a_n = 8^n - 5n$, при якому сума елементів прогресії перевищить 10000.

```
void main ()           // початок програми мовою c++
{
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 $y = \frac{20000}{8x^2 + 25}$ стане менше 20.

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

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

**Варіант 4.** Знайти ціле значення аргументу, при якому функція $y = \frac{7^x}{5x^2}$ перевищить 300.

**Варіант 5**. Знайти ціле значення аргументу, при якому функція $y = \frac{2000 + x}{8x^2 + 25}$ стане менше 10.

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

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

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

**Варіант 9**. Знайти ціле значення аргументу, при якому функція $y = \frac{300x}{8^x + 14}$ стане менше 5.

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

**Варіант 11**. Знайти ціле значення аргументу, при якому функція $y = \frac{40000}{8x^3 + 25}$ стане менше 14.

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

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

**Варіант 14.** Знайти ціле значення аргументу, при якому функція $y = \frac{4^x}{5x^2 - 4}$ перевищить 300.

**Варіант 15**. Знайти ціле значення аргументу, при якому функція $y = \frac{2000 + x}{8x^2 + 25 + 2}$ стане менше 10.

**Варіант 16**. Знайти ціле значення аргументу, при якому функція $y = \frac{9x^2 - 8x + 15}{x - 1}$ стане більше 1000.

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

**Варіант 18**. Знайти ціле значення аргументу, при якому функція $y=\frac{7x^2+25x-27}{x-1}$ стане більше 3000.

**Варіант 19**. Знайти ціле значення аргументу, при якому функція $y=\frac{300x+10}{7^x+14}$ стане менше 5.

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

**Варіант 21**. Знайти ціле значення аргументу, при якому функція $y=\frac{10000}{4x^2-15}$ стане менше 70.

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

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

**Варіант 24.** Знайти ціле значення аргументу, при якому функція $y=\frac{5^x+20}{5x^2-15}$ перевищить 300.

**Варіант 25**. Знайти ціле значення аргументу, при якому функція $y=\frac{2000+x}{4x^2-10}$ стане менше 2.

## Лабораторна робота № 11
## Організація циклів

1. У Microsoft Visual Studio створити проект консольного додатку (x86) 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-ми елементів. Помістити в регістр EAX максимальний елемент масиву, а в регістр ESI його адресу 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 за спаданням.

**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 $y = \frac{n!}{n}$ (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-ми елементів. Помістити в регістр EAX максимальний парний елемент масиву, а в регістр ESI його адресу в пам’яті.

**Варіант 16**. У пам’яті заданий масив з 9-ти елементів. Відсортувати елементи масиву за зростанням 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 за спаданням 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 адресу 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 IV
# ПРОГРАМУВАННЯ МАТЕМАТИЧНОГО СПІВПРОЦЕСОРА X86(FPU)

## 4.1 Формати чисел з плаваючою комою

Співпроцесор x87 розпізнає три формати чисел з плаваючою комою, що зберігаються в пам'яті (рис. 4.1). Усередині співпроцесора всі числа перетворюються в розширений формат.

Кожен формат складається з трьох полів: знак (S), порядок і мантиса (рис. 4.1). 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 4.1 – Формати чисел з плаваючою комою

Але якщо старший біт завжди містить 1, мантису можна не зберігати в кожному числі з плаваючою комою. 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 комою.

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 комою в пам'яті:

- одинарна точність: $(-1)^S (1 . X1\ X2 \ldots X23) \cdot 2^{(E-127)}$;
- подвійна точність: $(-1)^S (1 . X1\ X2 \ldots X52) \cdot 2^{(E-1023)}$;
- розширена точність: $(-1)^S (X1.\ X2 \ldots X64) \cdot 2^{(E-16383)}$,

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 [26].

Таблиця 4.1 – 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 комою в десятковій системі числення наведено в табл. 4.1 і на рис. 4.1.

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. 4.2 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 комою і навпаки. Це необхідно при обчисленнях, в яких є числа обох форматів. Допустимі формати цілих чисел наведені на рис. 4.3.

***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.

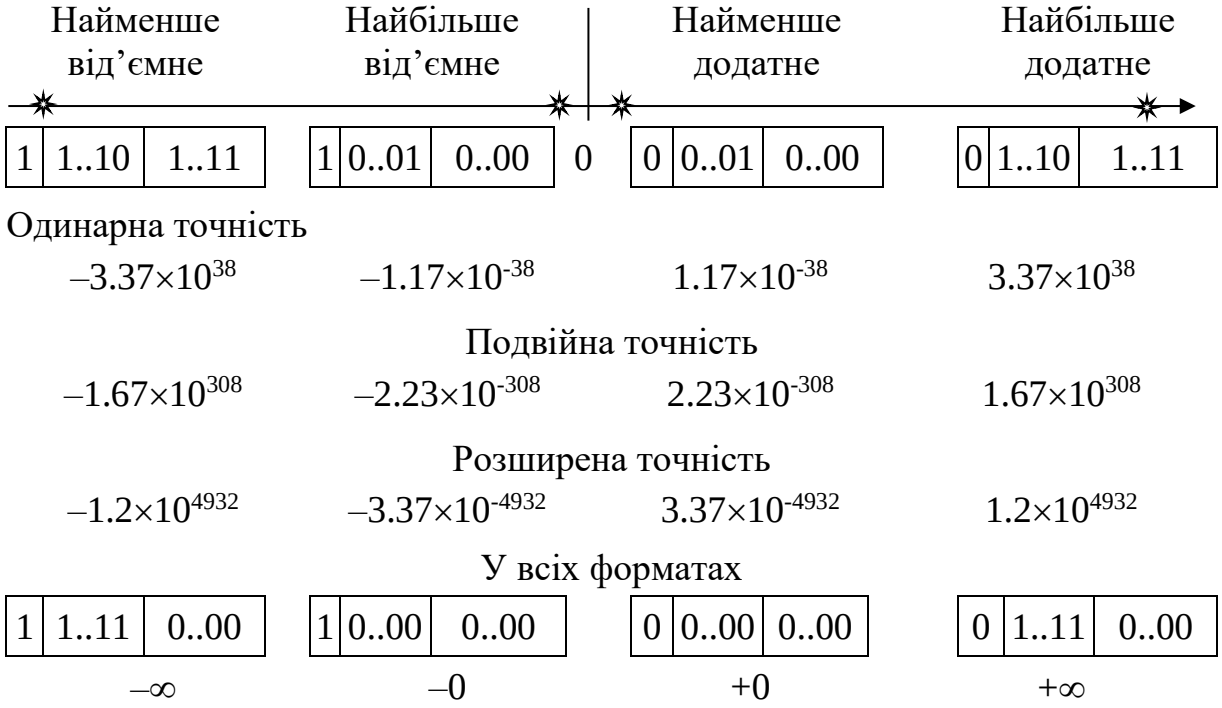



Рисунок 4.2 – Діапазон подання чисел

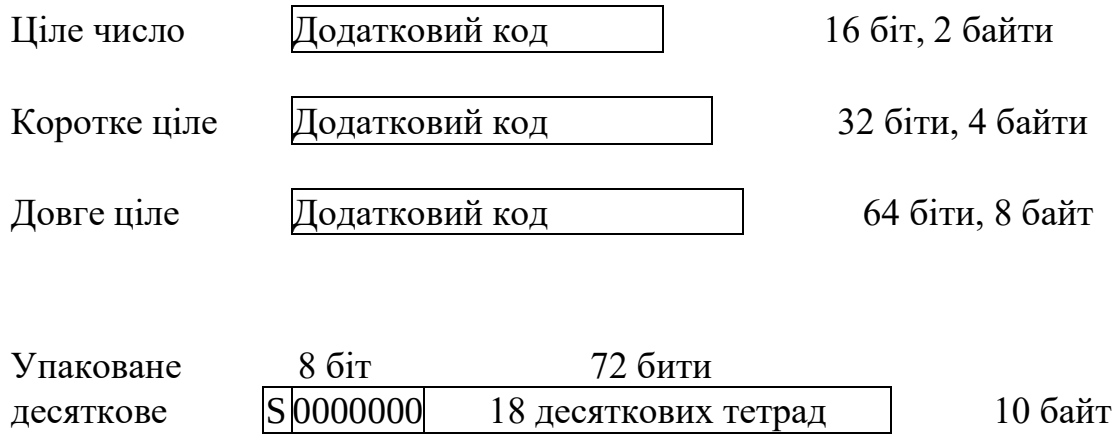



Рисунок 4.3 – Формати цілих чисел

При виконанні арифметичних операцій над числами з плаваючою комою 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 нескінченностей прихований (як прихований факт наявності двох нулів), тобто додатна нескінченність замикається на від'ємну нескінченність (порівняння двох нескінченностей 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 нескінченностями.

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

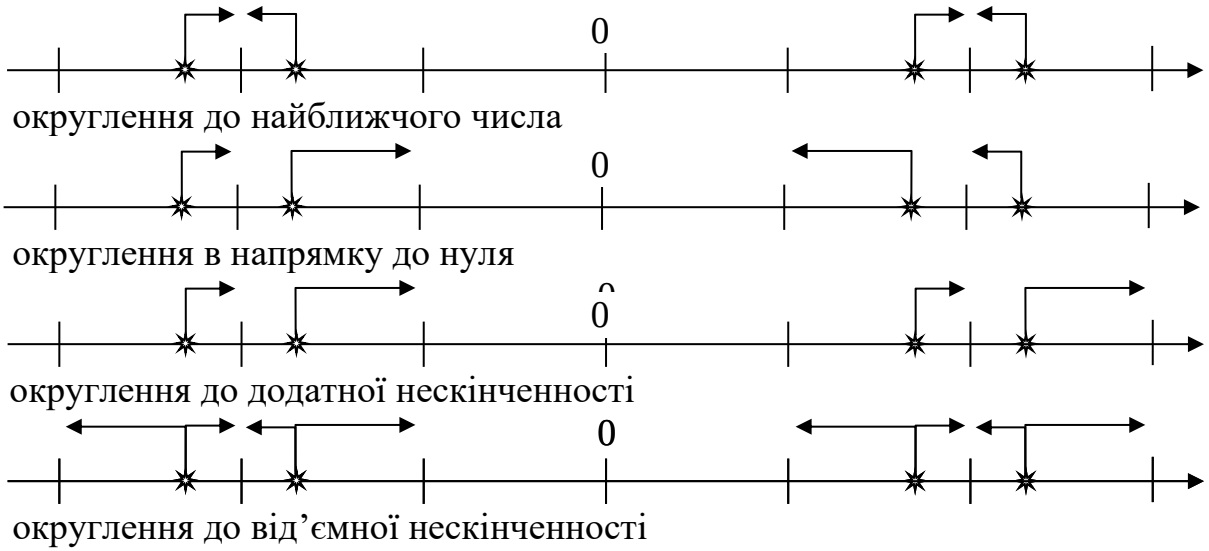



Рисунок 4.4 – Режими округлення неточного результату

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 комою. Програміст може вибирати один з чотирьох режимів округлення (результати операцій на рис. 4.4 подані зірочками) [26].

## 4.2 Регістри співпроцесора x87

Співпроцесор x87 має регістри (рис. 4.5), зручні для обчислень над числами з плаваючою комою.

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. 4.6).

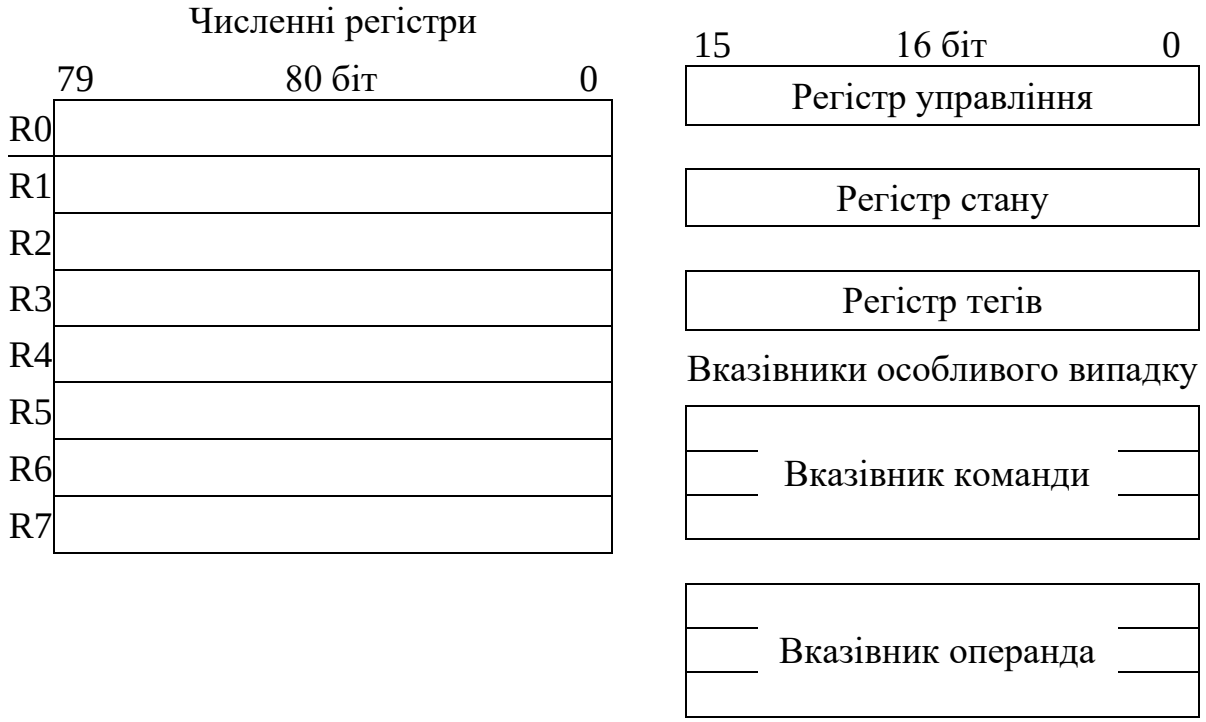



Рисунок 4.5 – Регістри співпроцесора x87

16-ти бітовий ***регістр управління*** містить поле ***масок особливих випадків*** і поля округлення чисел (рис. 4.8).

Молодші 6 бітів відведені для масок особливих випадків:

- IM – маска недійсної операції;
- DM – маска денормалізованного операнду;
- ZM – маска ділення на нуль;
- OM – маска переповнення;
- UM – маска антипереповнення;
- PM – маска точності (неточного результату).

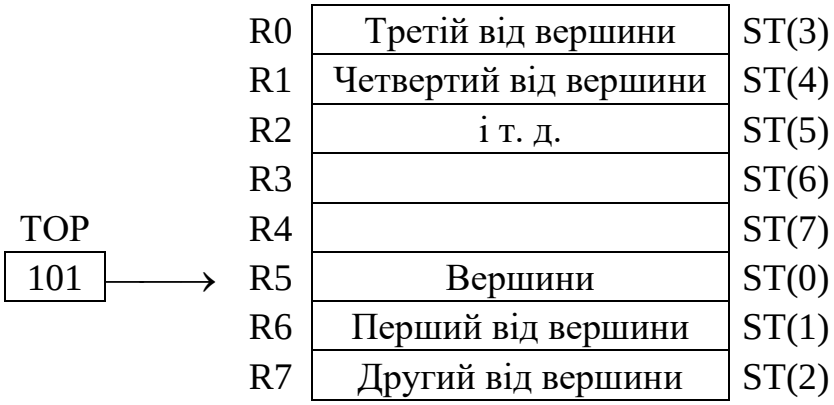



Рисунок 4.6 – Адресація численних регістрів як стека

16-ти бітовий ***регістр стану*** містить прапори, що модифікуються після виконання команд, а також поле вершини стека (TOP) (рис. 4.7).

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SW | B | C3 | TOP | | | C2 | C1 | C0 | ES | SF | PE | UE | OE | ZE | DE | IE |

Рисунок 4.7 – Регістр стану (Status Word)

Молодші 6 біт містять ***прапори особливих випадків***:

- IE – недійсна операція;
- DE – деморалізований 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 4.8 – 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. 4.9). 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.

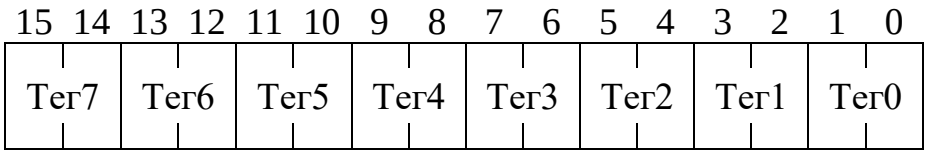



Рисунок 4.9 – Регістр тегів

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. 4.9).

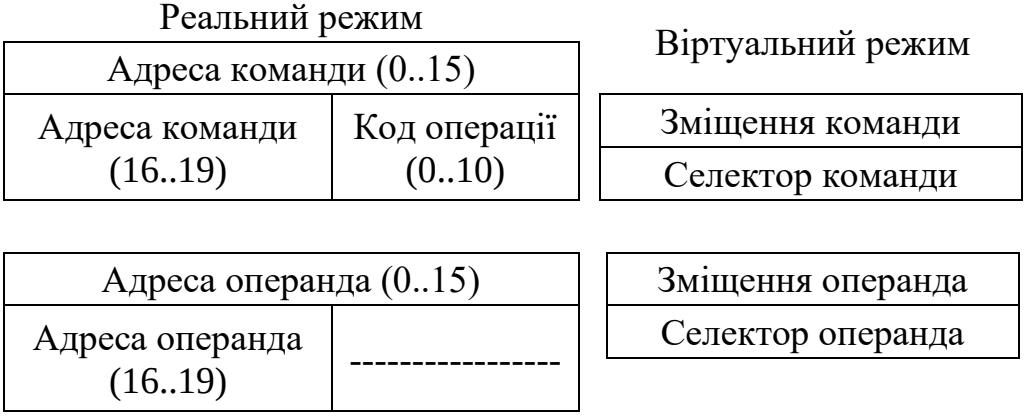



Рисунок 4.10 – Вказівники особливого випадку

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 [26].

## 4.3 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 комою).

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 [27].

**4.3.1 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 4.2 – Команди передачі даних

| Мнемоніка, операнд | | | Тип команди |
|---|---|---|---|
| З плаваючою комою | 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. 4.3.

Таблиця 4.3 – 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 комою 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. 4.11).

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. 4.12). Остання форма є частинним випадком попередньої [27].

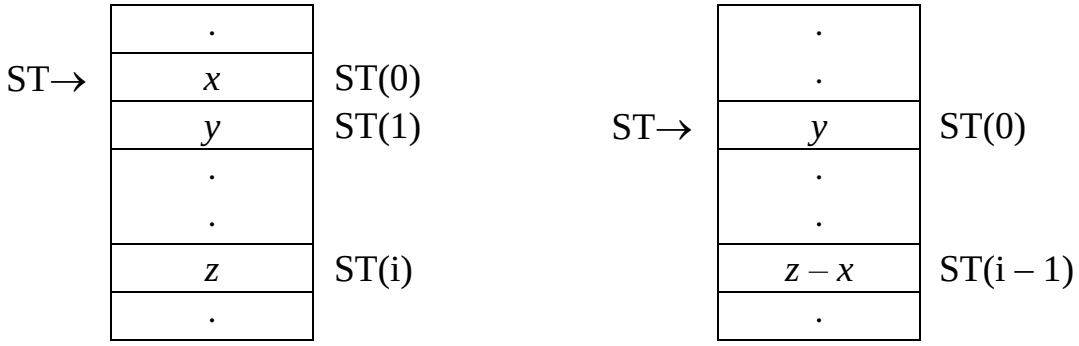



Рисунок 4.11 – Дія команди FSUBP ST(i),ST

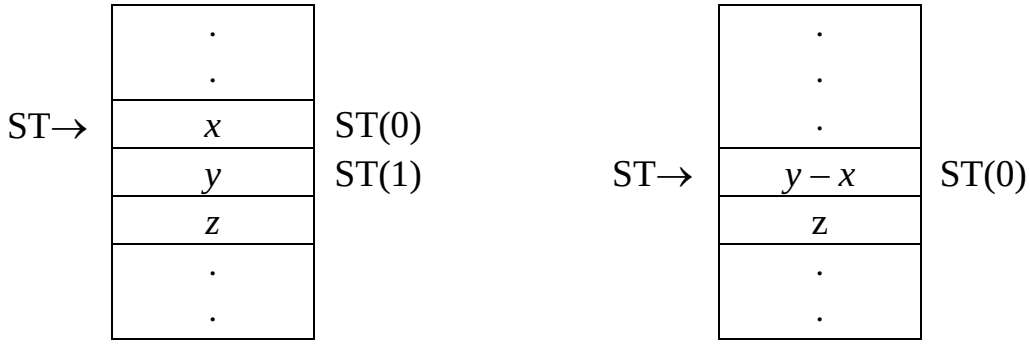



Рисунок 4.12 – Дія команди FSUB

***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 комою.

FSQRT – добування квадратного кореня.

FPREM – обчислення залишку 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. 4.4).

FICOM mem порівнює вмісти 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 операнду з вершини стека (P6 +).

Таблиця 4.4 – 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 [27].

***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));<br>ST(0) = cos (ST(0)) |
| FPATAN | Частковий арктангенс | ST(0) = arctg (ST(1)/ST(0) |
| FYL2X | Логарифм за основою 2 | $ST(0) = ST(1) \times \log_2 (ST(0))$ |
| FYL2XP1 | Логарифм за основою 2 | $ST(0) = ST(1) \times \log_2 (ST(0)+1)$ |
| 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):

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

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):

$$0 \le ST(0) \le 1/4 .$$

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

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

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 4.6 – Формули для обчислення тригонометричних функцій

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

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. 4.7 (***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. 4.8) із завантаженням в регістр ST(1) необхідних констант командами: FLDLN2 і FLDLG2.

Ще одна логарифмічна команда **FYL2XP1** обчислює функцію: $Y = \mathrm{ST}(1)\times\log_2(\mathrm{ST}(0)+1)$. 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:

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

Аргумент показникової функції повинен знаходиться в діапазоні:

- для співпроцесорів 87/287: $0 \le \mathrm{ST}(0) \le 0{,}5$;
- для співпроцесорів 387+: $-1 \le \mathrm{ST}(0) \le 1$.

Таблиця 4.7 – Формули для обчислення обернених тригонометричних функцій

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

Таблиця 4.8 – Формули для обчислення логарифмічних функцій

| |
|---|
| $\log_2 x \rightarrow$FLD1; FLD $x$; FYL2X; |
| $\ln x = \ln 2 \log_2 x \rightarrow$FLDLN2; FLD $x$; FYL2X; |
| $\lg x = \lg 2 \log_2 x \rightarrow$ FLDLG2; FLD $x$; FYL2X. |

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 4.9 – Формули для обчислення показникових функцій

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

Таблиця 4.10 – Формули для обчислення гіперболічних функцій

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

Таблиця 4.11 – Формули для обчислення обернених гіперболічних функцій

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

**4.3.2 Команди управління співпроцесором x87.** Команди управління співпроцесором x87 забезпечують доступ до нечислових регістрів. Мнемоніки, 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. 4.12. Таку ж дію здійснює апаратний сигнал скидання RESET.[27]

Таблиця 4.12 – Ініціалізація співпроцесора x87

| Регістр | Вибір | Режим роботи |
|---|---|---|
| Регістр управління | Режим нескінченності | Проективний – (287)<br>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 № 12
## Обчислення значення функції з використанням FPU

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання. Обчислити значення функції

$$S = \frac{1}{2} \cdot a \cdot b \cdot \sin\alpha$$

```
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 | Завдання | Варіант | Завдання |
|---|---|---|---|
| 1 | $y=3,5x^2+7,2x+24$ | 14 | $y=\sqrt{14x^2+5,6x-20}$ |
| 2 | $a_n=2,5n^2+5,3n-21$ | 15 | $y=\frac{20x-15}{5x^2-8,5x+15,2}$ |
| 3 | $y=\frac{2500}{2x^2+3,7}$ | 16 | $a_n=\frac{4,5^n-1}{n+5,4}$ |
| 4 | $y=\sqrt{14x^2+5,6}$ | 17 | $y=\frac{x-7}{\sqrt{x^2+20x-24}}$ |
| 5 | $y=\frac{20x}{5x^2-8,5}$ | 18 | $y=\frac{1024-2x^2}{3,2x^2-25}$ |
| 6 | $a_n=\frac{4,5^n}{n+5,1}$ | 19 | $a_n=\frac{418+n}{2n^2+2,3}$ |
| 7 | $y=\frac{x-7}{\sqrt{x^2+20}}$ | 20 | $y=\frac{3000-2x^2+2,5x}{x^2+3,6x-7,5}$ |
| 8 | $y=\frac{1024}{3,2x^2-25}$ | 21 | $y=\frac{3,5x^2+7,2x+24}{x^2-23,4}$ |
| 9 | $a_n=\frac{418}{2n^2+7,3}$ | 22 | $a_n=\frac{2,1n^2-2,3n-21}{n+1,4}$ |
| 10 | $y=\frac{3000+x}{x^2+2,6x-7,5}$ | 23 | $y=\frac{4000}{2x^2+3,7x-4,3}$ |
| 11 | $y=3,5x^2+7,2x-2,2$ | 24 | $y=\sqrt{14x^2+5,6x-2,4}$ |
| 12 | $a_n=2,5n^2+5,3n+5$ | 25 | $y=\frac{1124,4-x}{3,2x^2-25}$ |
| 13 | $y=\frac{2500,12}{2x^2+3,7x-8,1}$ | | |

## Лабораторна робота № 13
## Умовні переходи FPU

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання. Визначити номер ($n$) елемента прогресії $a_n=5,3^n+5n$, 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 $y = \frac{20}{x^2 + 2{,}5^x}$ стане меншою за 0,2.

**Варіант 2.** Знайти ціле значення аргументу $x$, при якому функція $y = \frac{15}{x^2 + 3{,}7}$ стане меншою за 0,1 (для $x$ – від 1 із кроком 1).

**Варіант 3.** Визначити номер ($n$) елемента прогресії $a_n = 2{,}5n^2 + 7{,}3$, 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 $a(i)=\sin(5i)$. Визначити номер елемента масиву, при якому сума елементів перевищить 3. Аргумент синуса заданий у градусах.

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

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

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

**Варіант 9.** Знайти ціле значення аргументу, при якому функція $y=\dfrac{5{,}6^x}{3x^2}$ перевищить 200.

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

**Варіант 11**. Знайти ціле значення аргументу, при якому функція $y=\dfrac{30}{1+x^2+2{,}5^x}$ стане менше 0,2.

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

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

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

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

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

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

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

**Варіант 19.** Знайти ціле значення аргументу, при якому функція $y=\dfrac{2{,}3+5{,}6^x}{2+3x^2}$ перевищить 400.

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

**Варіант 21**. Знайти ціле значення аргументу, при якому функція $y=\frac{40x-4,3}{x^2+2,5^x}$ стане менше 0,1.

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

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

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

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

## Лабораторна робота № 14
## Тригонометричні функції FPU

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання. Обчислити функцію $y=5\,arcsin\left(\frac{1}{7}(\text{tg}60°)^2\right)$. Результат перевести в градуси.

```
void main ()            // початок програми мовою З++
{
long  A=60, B=5, C=7;
long  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 С++
```

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.** $y = 3\arcsin\left(2\cos 70^\circ\right)^2$.

**Варіант 2.** $y = \frac{1}{2}\operatorname{arccosec}\left(3a^2 + 4b\right)$.

**Варіант 3.** $y = 2\arcsin\frac{a^2 + b^2}{3}$.

**Варіант 4.** $y = 3\arccos\left(2a^2 - b\right)$.

**Варіант 5.** $y = 5\operatorname{arcsec}\left(3\left(\operatorname{tg} 70^\circ\right)^2\right)$.

**Варіант 6.** $y = \frac{1}{3}\arcsin\left(3a + \sin 20^\circ\right)$.

**Варіант 7.** $y = 4\arccos\left(2\sin 30^\circ \cos 30^\circ\right)$.

**Варіант 8.** $y = 3\operatorname{arccosec}\frac{5a + b^2}{2}$.

**Варіант 9.** $y = 3\operatorname{arccosec}\left(4a + \operatorname{tg} 40^\circ\right)$.

**Варіант 10.** $y = 5\arcsin\left(3\operatorname{tg} 25^\circ \sin 25^\circ\right)$.

**Варіант 11.** $y = 3\arcsin\left(2\sin 70^\circ\right)^2$.

**Варіант 12.** $y = \frac{1}{4}\operatorname{arcsec}\left(3a^2 + 4b\right)$.

**Варіант 13.** $y = 4\arccos\frac{a^2 + b^2}{3}$.

**Варіант 14.** $y = 3\arcsin\left(2a^2 - b\right)$.

**Варіант 15.** $y = 5\,\mathrm{arcsec}\left(3\left(\sin 70^\circ\right)^2\right)$.

**Варіант 16.** $y = \frac{1}{3}\arcsin\left(3a + \sin 20^\circ\right)$.

**Варіант 17.** $y = 2\,\mathrm{arctg}\left(2\sin 30^\circ \cos 30^\circ\right)$.

**Варіант 18.** $y = \frac{1}{4}\,\mathrm{arccosec}\frac{5a + b^2}{2}$.

**Варіант 19.** $y = 1{,}5\,\mathrm{arcsec}\left(4a + \mathrm{tg}30^\circ\right)$.

**Варіант 20.** $y = 2\arccos\left(3\cos 25^\circ \sin 25^\circ\right)$.

**Варіант 21.** $y = 3{,}4\arcsin\left(2\cos 40^\circ\right)^2$.

**Варіант 22.** $y = \frac{1}{3}\,\mathrm{arccosec}\left(1{,}2a^2 + 4b\right)$.

**Варіант 23.** $y = 2a\arccos\frac{a^2 + b^2}{3}$.

**Варіант 24.** $y = 3{,}8\arcsin\left(2a^2 - b - c\right)$.

**Варіант 25.** $y = 5{,}1\,\mathrm{arcsec}\left(3\left(\sin 70^\circ\right)^2\right)$.

## Лабораторна робота № 15
## Логарифмічні та показникові функції FPU

1. У Microsoft Visual Studio створити проект консольного додатку (x86) на C++.
2. Ввести наведений нижче код прикладу.
3. Приклад виконання завдання. Піднесення основи 2 до довільного степеню. У вершині стеку знаходиться аргумент «n». Результат ($2^n$) розмістити у вершині стеку. Фрагмент програми:

```
fld ST(0)          ;копіювання вершини стеку
frndint            ;округлення ST(0)до цілого
fsub ST(1),ST(0)   ;виділення дробової частини в ST(1)
fxch               ;обмін регістрів ST(0), ST(1)
f2xm1              ;піднесення 2 до дробового степеню (мінус 1)
fld1               ;завантаження константи «1»
fadd               ;додавання одиниці
fscale             ;піднесення 2 до цілого степеню та множення
fstp ST(1)         ;видалення регістру та зсув вершини стеку.
                   ;результат в вершині стеку
```

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 $y = 4\lg \mathrm{tg} x$, $x$ 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 $y = 3\log_8\left(x^2 + 1\right)$, $x$ 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 $y = 7\ln\left(x^2 + \sqrt{x}\right)$, $x$ змінюється від 2 із кроком 3.

**Варіант 8.** Обчислити 5 значень функції $y = 4\left(x^2 + 1\right)$, $x$ 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 $y = 3\ln\left(4\sin 2x\right)$, $x$ 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 $y = 4\lg \mathrm{ctg} 2x$, $x$ 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 $y = 3\log_8\left(x^2+1\right)$, $x$ змінюється від 0,2 із кроком 0,3.

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

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

**Варіант 18.** Обчислити 5 значень функції $y = \ln\dfrac{4\left(x^2+1\right)}{x-2}$, $x$ 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 $y = 3^{\cos(2-5{,}1x)}$, $x$ 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 $y = \ln\left(25{,}4x+7^x\right)$, $x$ змінюється від 0,5 із кроком 0,2.

**Варіант 23.** Обчислити 7 значень функції $y = 2\ln\operatorname{ctg} x - 10{,}5$, $x$ змінюється в градусах від 12 із кроком 10.

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

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

# ЧАСТИНА V
# ПРОГРАМУВАННЯ ПРОЦЕСОРА INTEL 64

Процесори архітектури Intel 64 підтримують два режими роботи: Long mode («довгий» режим) та Legacy mode («успадкований», режим сумісності з 32-бітним x86).

**Long mode** («довгий» режим) – основний для процесорів Intel 64. Цей режим дає можливість скористатися всіма додатковими перевагами, які надає архітектура. Для використання цього режиму потрібна 64-бітна операційна система.

Цей режим дозволяє виконувати 64-бітові програми; також (для зворотної сумісності) надається підтримка виконання 32-бітного коду, наприклад, 32-бітових програм, хоча 32-бітові програми не зможуть використовувати 64-бітові системні бібліотеки, і навпаки. Щоб вирішити цю проблему, більшість 64-розрядних операційних систем надають два набори необхідних системних файлів: один – для рідних 64-бітних програм, а інший – для 32-бітових програм.

**Legacy mode** – «успадкований» режим дозволяє процесору виконувати інструкції, розраховані для процесорів x86, і надає повну сумісність із 32-бітним кодом та операційними системами. У цьому режимі процесор поводиться так само, як x86-процесор, наприклад Athlon або Pentium III, і додаткові функції, що надаються архітектурою Intel 64 (наприклад, додаткові регістри), недоступні. У цьому режимі 64-бітові програми та операційні системи не працюватимуть.

Цей режим включає підрежими:

- реальний режим (real mode);
- захищений режим (protected mode);
- режим віртуального 8086 (virtual 8086 mode) (рис 5.1) [28].

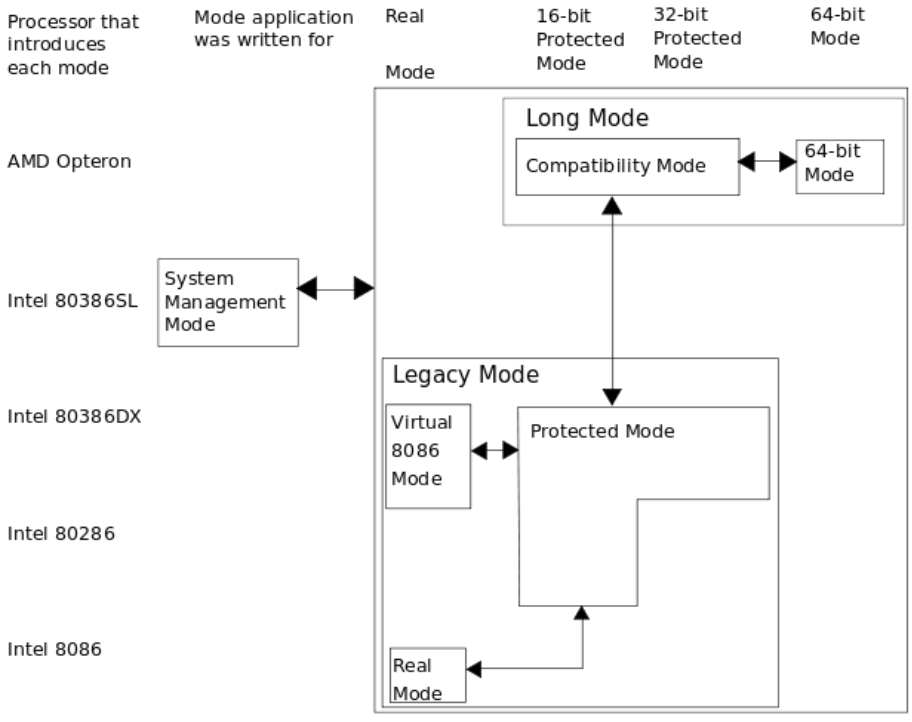



Рисунок 5.1 – Режими роботи процесора архітектури Intel 64

## 5.1 Програмна модель 64-х розрядних процесорів

В 64 розрядному режимі доступні наступні регістри:

- регістри загального призначення: 64-розрядні RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP та R8, R9, ... R15; 32-розрядні EAX, ЄBX, ЄCX, EDX, ESI, EDI, ESP, EBP, R8D-R15D (є молодшими частинами 64-розрядних регістрів); 16-розрядні AX, BX, СГ, DX, SI, DI, SP, BP, R8W-R15W (є молодшими частинами 32-розрядних регістрів); 8-бітові регістри AH, BH, CH, DH та AL, BL, CL, DL, SIL, DIL, SPL, BPL, R8L-R15L (старші та молодші частини 16-бітних регістрів відповідно) (рис. 5.1);
- 16-розрядні сегментні регістри: CS, DS, SS, ES, FS, GS (рис 5.2);
- 64-розрядний RIP – покажчик інструкції (рис 5.3);
- 64-розрядний регістр прапорів – RFLAGS (рис 5.3);
- 80-бітові регістри математичного співпроцесора ST0-ST7;
- 64-бітові MMX-реєстри (MMO-MM7);
- 128-розрядні XMM-реєстри – XMMO-XMM15 та 32-бітний MXCSR;
- 64-розрядні регістри управління CR0-CR4 та CR8;
- регістри-покажчики системних таблиць GDTR, LDTR, IDTR та регістр задачі TR; 64-розрядні регістри налагодження – DR0-DR3, DR6, DR7;
- MSR-реєстри [28].

**Регістри загального призначення**

| 63 | 31 | 15 8 | 7 0 | |
|---|---|---|---|---|
| | EAX | AH | AX AL | RAX |
| | EBX | BH | BX BL | RBX |
| | ECX | CH | CX CL | RCX |
| | EDX | DH | DX DL | RDX |
| | ESI | SI | | RSI |
| | EDI | DI | | RDI |
| | EBP | BP | | RBP |
| | ESP | SP | | RSP |
| | R8D | R8W | R8L | R8 |
| | R9D | R9W | R9L | R9 |
| | R10D | R10W | R10L | R10 |
| | R11D | R11W | R11L | R11 |
| | R12D | R12W | R12L | R12 |
| | R13D | R13W | R13L | R13 |
| | R14D | R14W | R14L | R14 |
| | R15D | R15W | R15L | R15 |

Рисунок 5.1 – Регістри загального призначення

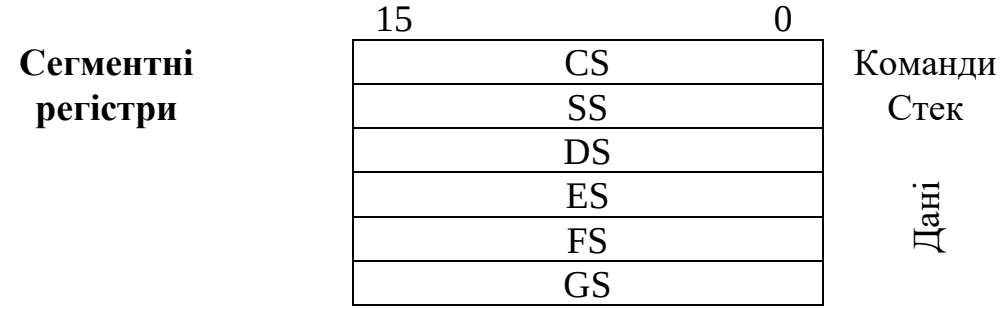



Рисунок 5.2 – Сегментні регістри

**Вказівник команд і регістр прапорів**

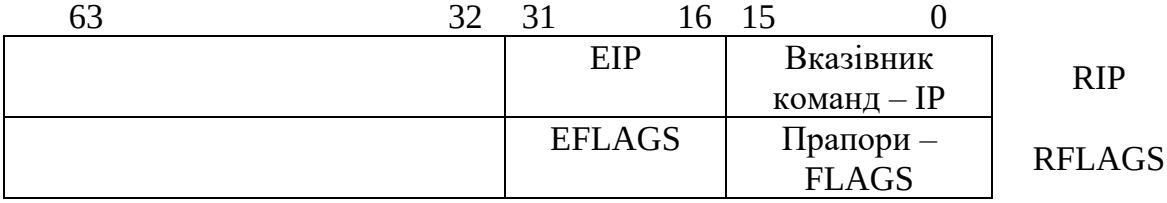



Рисунок 5.3 – Вказівник команд та регістр прапорів

Верхні 32 біти регістру RFLAGS зарезервовані, а нижні відповідають 32-бітному регістру EFLAGS, який розглянуто у III частині цього посібника.

Операції 64-бітних регістрів (додавання, віднімання та ін.) працюють стільки ж часу, скільки операції 32-бітних регістрів.

Може здатися дивним, але операції над молодшими 32-бітними половинами регістрів обнуляють старші 32 біти. Наприклад, mov eax, ebx автоматично обнуляє старші біти в rax. Так зроблено для оптимізації. Це дозволяє процесору розривати ланцюжки залежностей.

В архітектурі Intel64 у 64-бітному режимі сегментація не використовується. Для чотирьох сегментних регістрів (CS, SS, DS і ES) базова адреса примусово виставляється в 0. Сегментні регістри FS і GS, як і раніше, можуть мати ненульову базову адресу. Це дозволяє ОС використовувати їх для службових цілей.

Наприклад, Microsoft Windows X64 використовує GS для вказівки на Thread Environment Block, маленьку структурку для кожного потоку, яка містить інформацію про обробку винятків, thread-local-змінних та інших per-thread-відомостей. Аналогічно ядро Linux використовує GS-сегмент для зберігання даних per-CPU.

Режими адресації в 64-розрядному режимі схожі на x86, але не ідентичні.

Інструкції, що посилаються на 64-розрядні регістри, автоматично виконуються з 64-розрядною точністю. Наприклад, mov rax, [rbx] переміщає 8 байт, починаючи з rbx в rax.

Для 64-розрядних безпосередніх констант або адрес констант додано спеціальну форму інструкції mov. Для всіх інших інструкцій безпосередні константи або адреси констант, як і раніше, є 32 бітами.

x64 надає новий режим адресації щодо rip. Інструкції, які посилаються однією адресу константи, кодуються як зміщення від rip. Наприклад, інструкція mov rax, [addr] переміщає 8 байт, починаючи з addr + rip в rax.

Інструкції, такі як jmp, call, push і pop, які неявно посилаються на покажчик інструкції, покажчик стека обробляє як 64-розрядні регістри x64 [28].

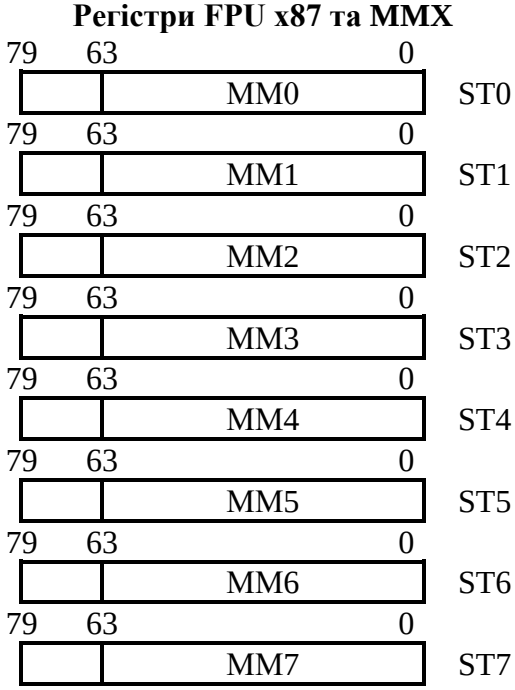



Рисунок 5.4 – Регістри FPU x87 та MMX

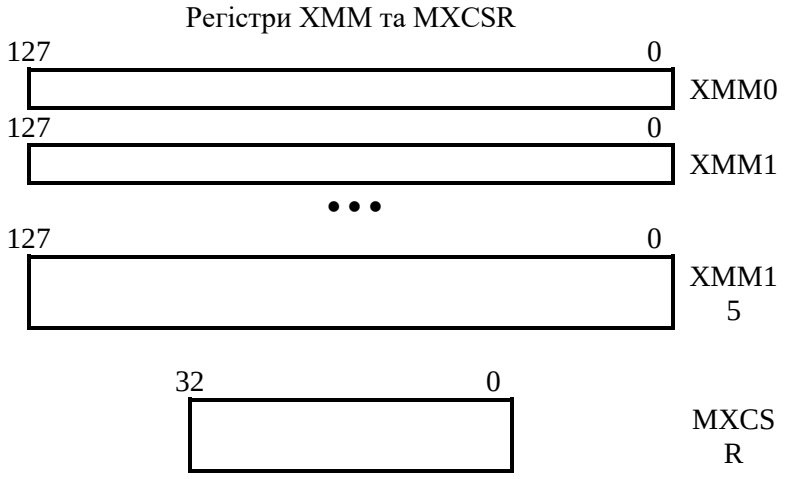



Рисунок 5.5 – Регістри XMM

До системних адресних регістрів відносяться чотири (16-бітові) регістри таблиць GDTR, IDTR, LDTR, TR.

- GDTR – регістр, в якому міститься лінійна адреса глобальної дескрипторної таблиці;
- IDTR – регістр, який містить 32-бітову лінійну адресу таблиці дескрипторів обробників переривань;

- LDTR – регістр, який містить 16-бітовий селектор (індекс) для GTD і 8-байтовий дескриптор;
- TR – регістр, який містить 16-бітовий селектор (індекс) для GTD і 8-байтовий дескриптор з GTD, який описує TSS поточної задачі.

До керуючих відносяться чотири регістри CR0, CR2, CR3, CR4, CR8.

Регістр CR0:

- 0-біт, дозвіл захисту. Переводить процесор у захищений режим;
- 1-біт, моніторинг співпроцесора (mp). Викликає виняток 7 для кожної команди wait;
- 2-біт, емуляція співпроцесора (em). Викликає виняток 7 для кожної команди співпроцесора;
- 3-біт, перемикання задач (ts). Дозволяє визначити, чи відноситься даний контекст співпроцесора до поточної задачі чи ні. Викликає виняток 7 при виконанні наступної команди співпроцесора;
- 4-біт, індикатор підтримки інструкцій співпроцесора (et);

Регістр CR2 зберігає 32-бітну лінійну адресу, для якої була отримана остання відмова сторінки пам'яті.

Регістр CR3:

- 3-біт, кешування сторінок із наскрізним записом (PWT);
- 4-біт, заборона кешування сторінки (PCD);
- 11-31, 20 старших бітів фізичної адреси таблиць каталогу сторінок при умові, що 5-й біт регістра CR4 дорівнює 1.

Керуючі регістри та регістри вказівників системної таблиці

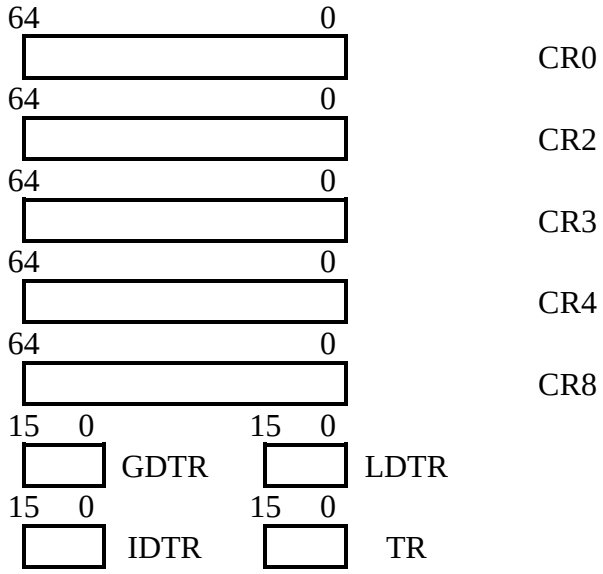



Рисунок 5.6 – Керуючі регістри та регістри вказівників системної таблиці

Регістр CR4:

- 0-біт, дозвіл використання віртуального прапора переривань в режимі V8086 (VME);

- 1-біт, дозвіл використання віртуального прапора переривань в захищеному режимі (VPI);
- 2-біт, перетворення інструкції RDTSC в привілейовану (TSD);
- 3-біт, дозвіл точок зупинки при зверненні до портів введення-виведення (DE);
- 4-біт, включає режим адресації з 4 мегабітними сторінками (PSE);
- 5-біт, включає 36-бітний фізичний адресний простір (PAE);
- 6-біт, дозвіл винятку MC (MCE);
- 7-біт, дозвіл глобальної сторінки (PGE);
- 8-біт, дозвіл виконання команди RDPMC (PMC);
- 9-біт, дозволяє команди швидкого збереження/відновлення стану співпроцесора (FSR) (рис 5.6) [28].

DR0-DR3 – зберігають 64-бітові лінійні адреси точок зупинки.

DR6 (еквівалентно DR4) – відображає стан контрольних точок.

DR7 (еквівалентно DR5) – керує встановленням контрольних точок (рис 5.7).

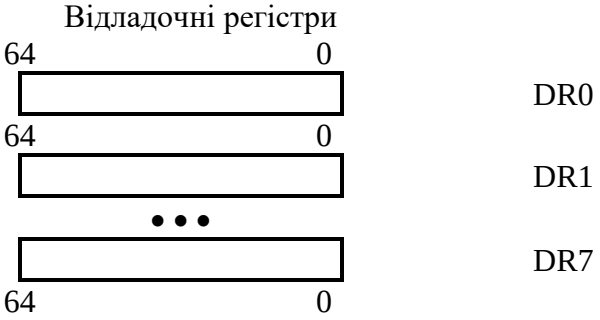



Рисунок 5.7 – Відладочні регістри

## 5.2 Типи даних 64-х бітових процесорів

Процесор в 64-х бітному режимі працює з цілими двійковими числами довжиною 8, 16, 32, 64 біта та числами з плаваючою точкою довжиною 16, 32, 64, 80 біт.

У двійкових цілих числах без знака всі розряди вважаються значущими (рис. 5.8). Двійкові цілі числа зі знаком подаються у додатковому коді. Старший біт є знаковим (рис. 5.8): S = 0 – число додатне, S = 1 – число від’ємне.

Вказівники у 64-розрядному режимі (підрежим режиму IA-32e) ближній вказівник має 64 біти. Це означає ефективну адресу. Далекі вказівники в 64-розрядному режимі можуть мати одну з трьох форм:

- 16-бітний селектор сегментів, 16-бітове зміщення, якщо розмір операнда становить 32 біти;
- 16-бітний селектор сегментів, 32-бітове зміщення, якщо розмір операнда становить 32 біти;

16-бітний селектор сегментів, 64-бітове зміщення, якщо розмір операнда становить 64 біти (рис. 5.9).

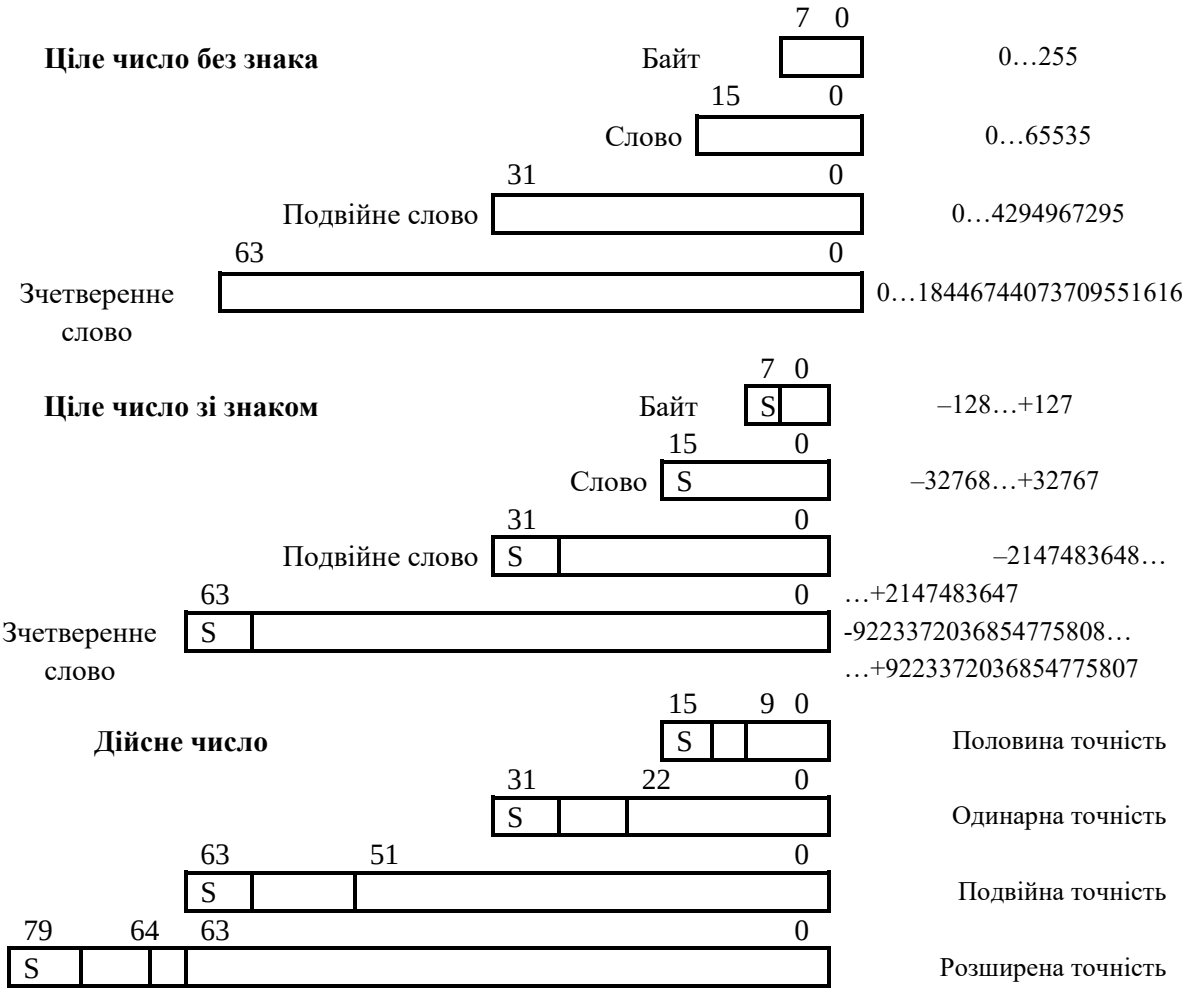



Рисунок 5.8 – Числові типи даних в 64-х бітному режимі

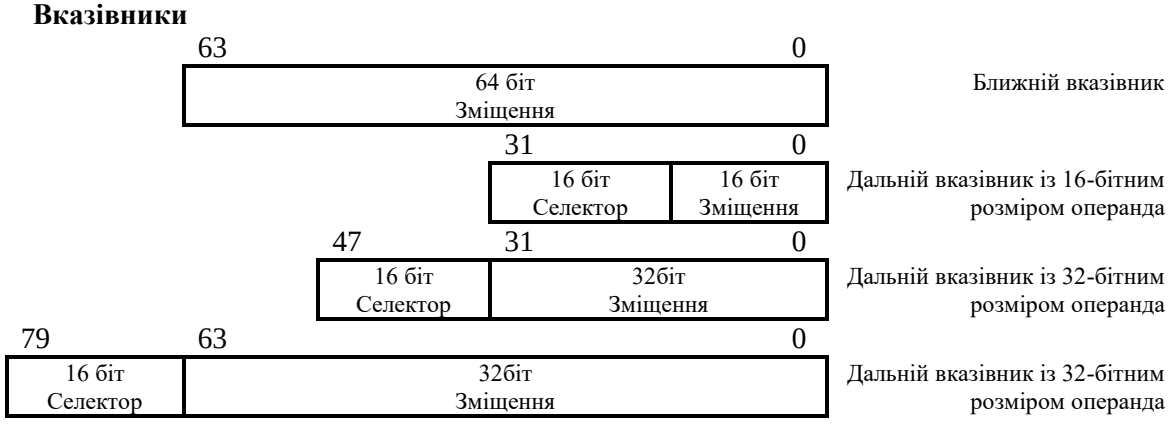



Рисунок 5.9 – Вказівники 64-х бітному режимі

64-розрядні упаковані типи даних SIMD були введені в архітектуру IA-32 у технології Intel MMX. Вони обробляються в регістрах MMX. Основними 64-розрядними типами упакованих даних є упаковані байти, упаковані слова та упаковані подвійні слова (рис 5.10). При виконанні числових операцій SIMD над цими типами даних вони інтерпретуються як такі, що містять цілі числа байт, слово або подвійне слово.

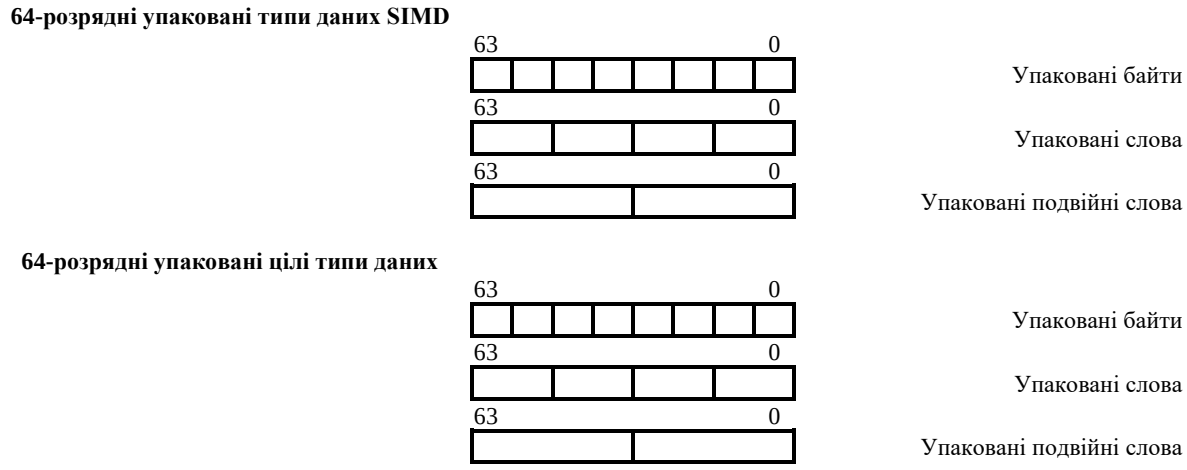



Рисунок 5.10 – Основні 64-розрядні упаковані типи даних SIMD

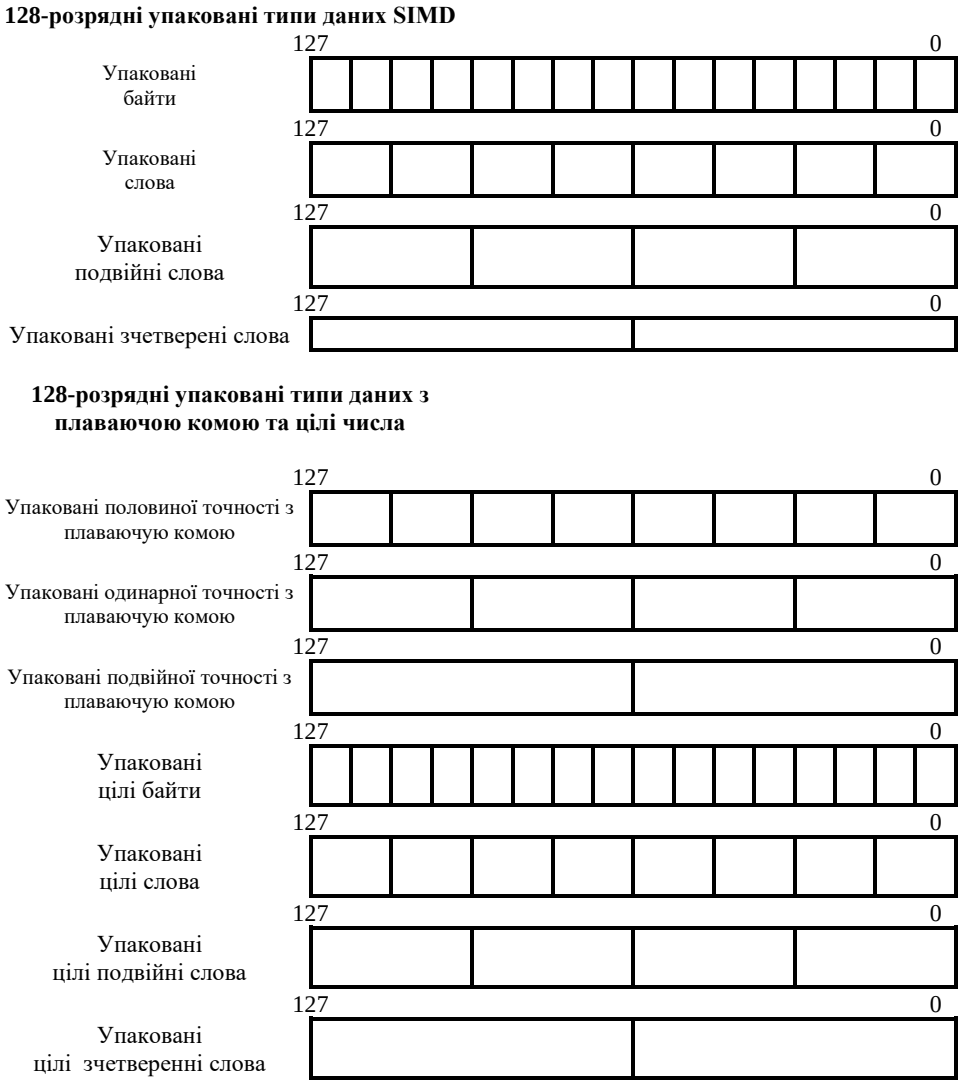



Рисунок 5.11 – 128-розрядні упаковані типи даних SIMD

128-розрядні упаковані типи даних SIMD були введені в архітектуру IA-32 у розширеннях Intel SSE та використовується з розширеннями Intel SSE2, SSE3, SSSE3, SSE4.1 і AVX. Вони працюють переважно в 128-розрядному режимі.

**Регістри та пам'ять XMM.** Основні 128-розрядні типи упакованих даних: упаковані байти, упаковані слова, упаковані подвійні слова та упаковані зчетверені слова (рис 5.11). При виконанні операцій SIMD над цими фундаментальними типами даних у регістрах XMM, вони інтерпретуються як такі, що містять упаковані або скалярні значення з плаваючою комою половинної точності, одинарної точності з плаваючою комою або подвійної точності з плаваючою комою, або як такі, що містять упаковані байти, слова, подвійне слово або цілі числа з чотирьох слів [28].

## 5.3 Програмування на асемблері в 64-х бітному режимі

**Увага !!! Microsoft Visual Studio не підтримує асемблерні вставки 64-х бітному режимі _asm{}!!!**

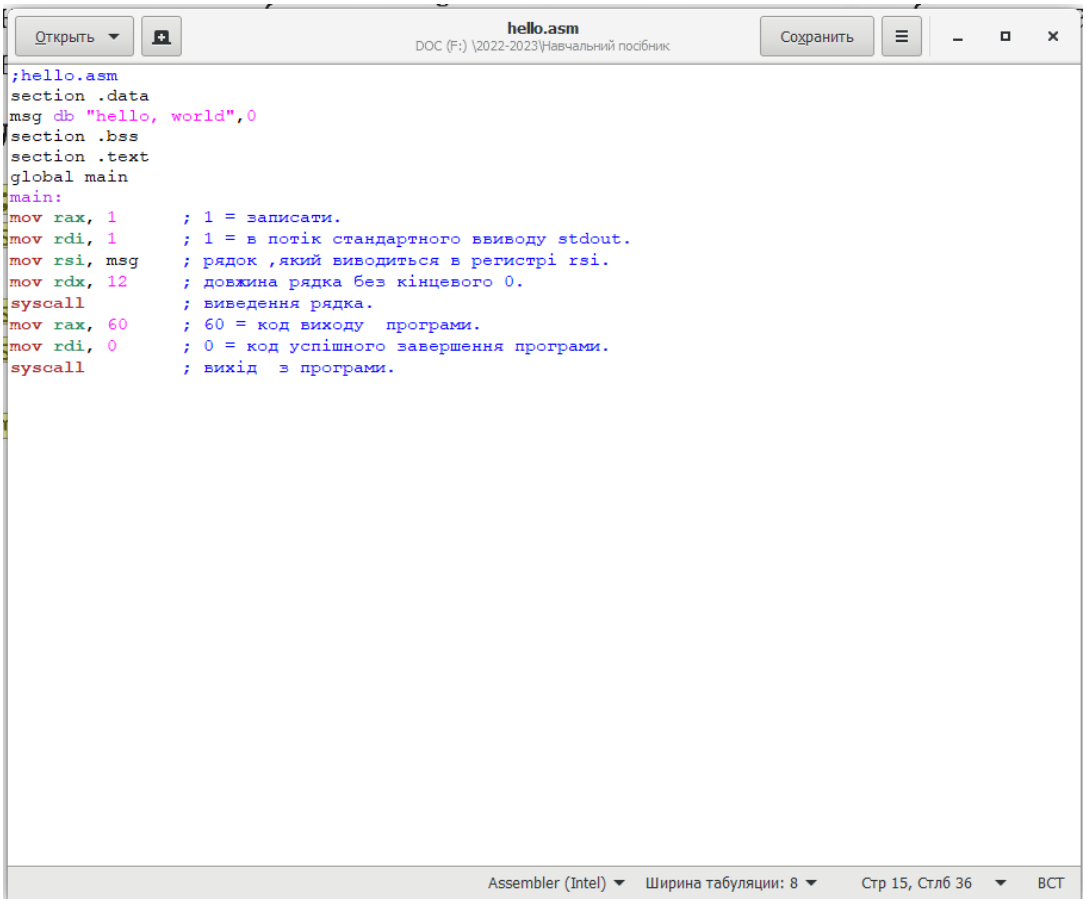



Рисунок 5.12 – Редактор gedit

Тут будемо розглядати код для версії компілятора Netwide Assembler (NASM), оскільки він доступний для ОС Linux, Windows. Довідку по NASM можна знайти тут: www.nasm.us. В якості редактора будемо використовувати gedit (https://gedit.en.uptodown.com/windows/download). Файл (модуль) підтримки підсвітки синтаксису можна знайти тут:

https://wiki.gnome.org/action/show/Projects/GtkSourceView/LanguageDefinitions. Завантажте файл asm-intel.lang, скопіюйте його в c:\Program Files\gedit\share\gtksourceview-3.0\language-specs\ для Windows, /usr/share/gtksourceview-3.0/language-specs/ для Linux/ При першому запуску gedit можна вибрати підтримувану мову програмування, у нашому випадку Assembler (Intel), в останній частині вікна gedit (рис. 5.12).

Надалі будемо розглядати приклади в ОС Linux.

Для програмування в Linux необхідно встановити наступні інструменти: GCC, GDB, make, NASM.

```
sudo apt install gcc gdb make
sudo apt install build-essential nasm
```

```
;hello.asm
section .data
msg db "hello, world",0
section .bss
section .text
global main
main:
mov rax, 1
mov rdi, 1
mov rsi, msg
mov rdx, 12
syscall
mov rax, 60
mov rdi, 0
syscall
```

Рисунок 5.13 – Вихідний код першої програми

Файл makefile буде використовуватися командою make для автоматизації збирання програми. Збирання (building) означає перевірку вихідного коду на помилки, додавання всіх необхідних сервісів операційної системи та перетворення вихідного коду на послідовність інструкцій, що розпізнаються комп'ютером. Дізнатися більше про файли makefile, можна у посібнику: https://www.gnu.org/software/make/manual/make.html.

```
#makefile for hello.asm
hello: hello.o
	gcc -o hello hello.o
hello.o: hello.asm
	nasm -f elf64 -g -F dwarf hello.asm -l hello.lst
```

Рисунок -5.14 makefile проекту

Результат виконання програми на ОС Kali Linux 2022.3 for VirtualBox (рис 5.15):

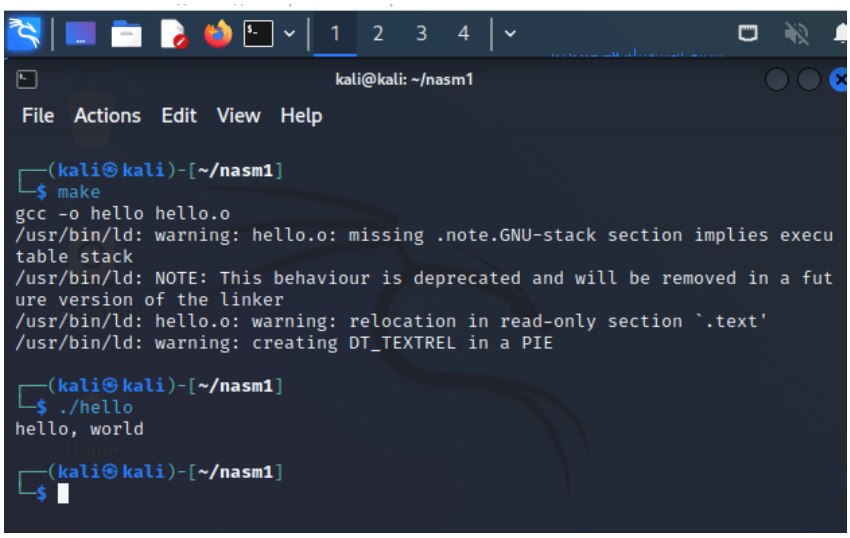



Рисунок 5.15 – Результат виконання програми

Перша програма демонструє базову структуру будь-якої асемблерної програми NASM. Нижче перераховані основні частини програми на асемблері:

- section .data;
- section .bss;
- section .txt.

У розділі section .data оголошуються та визначаються ініціалізовані дані ні в наступному форматі: <variable name> <type> <value>

Якщо змінна включена до розділу section .data, для неї виділяється пам’ять при асемблюванні та зв’язуванні вихідного коду для створення виконуваного коду. Змінні мають символьні імена та посилання на локації в пам’яті, при цьому змінна може займати одну або кілька комірок пам’яті.

Ім’я змінної означає початкову адресу змінної пам’яті.

Імена змінних повинні починатися з літери, за якою слідують літери, цифри або деякі спеціальні символи. У табл. 5.1 перераховані можливі типи даних.

Таблиця 5.1 – Типи даних .data

| Тип | Довжина | Назва |
|---|---|---|
| db | 8 біт | Байт |
| dw | 16 біт | Слово |
| dd | 32 біти | Подвійне слово |
| dq | 64 біти | Зчетверенне слово |

Розділ **section.data** може містити константи, тобто значення, які неможливо змінити у програмі. Константи визначаються в наступному форматі: <constant name> equ <value>

Приклад:

```
pi equ 3.1416
```

Розділ section .bss. Абревіатура bss означає Block Started by Symbol (блок, що починається з символу). У цьому розділі містяться неініціалізовані змінні. Для таких неініціалізованих змінних простір пам’яті, що виділяється, оголошується в наступному форматі: <variable name> <type> <number>

В таблиці. 5.2 наведено можливі типи даних bss.

Таблиця 5.2 – Типи даних .bss

| Тип | Довжина | Назва |
|---|---|---|
| resb | 8 біт | Байт |
| resw | 16 біт | Слово |
| resd | 32 біти | Подвійне слово |
| resq | 64 біти | Зчетверенне слово |

Наприклад, наступна інструкція оголошує простір пам’яті для масиву з 20 подвійних слів: dArray resd 20.

Змінні розділу **section .bss** не містять жодних значень, значення будуть присвоюватися їм надалі під час виконання програми. Блоки пам’яті цих змінних резервуються не під час компіляції, а під час виконання. Коли програма починає виконуватися, вона запитує в операційної системи необхідну пам’ять, що виділяється змінним з розділу section .bss та ініціалізується нулями. Якщо під час виконання немає доступної пам’яті, достатньої для розміщення змінних .bss, то програма завершується аварійно.

Розділ section .txt. Усі дії відбуваються у розділі section .txt. Цей розділ містить код програми та починається з наступних інструкцій:

**global main**

**main:**

Частина main: називається міткою (label). Якщо мітка розташована в рядку, де після неї немає інших символів, то після слова має бути записана двокрапка, інакше асемблер виведе попереджувальне повідомлення. А попереджувальні повідомлення не слід ігнорувати. Якщо за міткою слідують інші інструкції (у тому ж рядку), то двокрапка не обов’язково, але все ж таки краще виробити корисну звичку завершувати всі мітки символом двокрапки. До того ж, це підвищує зручність читання вихідного коду.

У прикладі (рис. 5.12), після мітки main: регістри rdi, rsi та rax готуються для виведення повідомлення на екран з використанням системного виклику.

Приклад вихідного коду (alive.asm), який демонструє, як можна визначити довжину рядка та як числові значення зберігаються у пам’яті.

Лістинг 5.1 – Файл alive.asm

```
;alive.asm
section .data
      msg1 db "Hello, World!",10,0        ;рядок NL та 0.
      msg1Len equ $-msg1-1                ;довжина  без 0.
      msg2 db "Alive and Kicking!",10,0   ;рядок з NL и 0.
      msg2Len equ $-msg2-1                ;довжина без 0
      radius dq 357                       ;це не рядок, чи можна вивести?
      pi dq 3.14                          ;це не рядок, чи можна вивести?
```

```
section .bss
section .text
global main
main:
      push rbp                              ;пролог функції.
      mov rbp, rsp                          ;пролог функції.
      mov rax, 1                            ;1=запис
      mov rdi, 1                      ;1 = в пристрій стандартного виведення stdout.
      mov rsi, msg1                         ;виведення рядка.
      mov rdx, msg1Len                      ;довжина рядка.
      syscall                               ;довжина рядка.
      mov rax, 1                            ;1 = запис
      mov rdi, 1                      ;1 = в пристрій стандартного виведення stdout.
      mov rsi, msg2                         ;виведення рядка.
      mov rdx, msg2Len                      ;довжина рядка
      syscall                               ;виведення рядка.
      mov rsp,rbp                           ;епілог функції.
      pop rbp                               ;епілог функції.
      mov rax, 60                           ;60=вихід
      mov rdi, 0                            ;0 = код успішного завершення.
      syscall                               ;вихід.
```

makefile для прикладу alive.asm.

```
#makefile for alive.asm
      alive: alive.o
            gcc -o alive alive.o
      alive.o: alive.asm
            nasm -f elf64 -g -F dwarf alive.asm -l alive.lst
```

Результат виконання проєкту alive.asm з використанням SASM (рис 5.16)

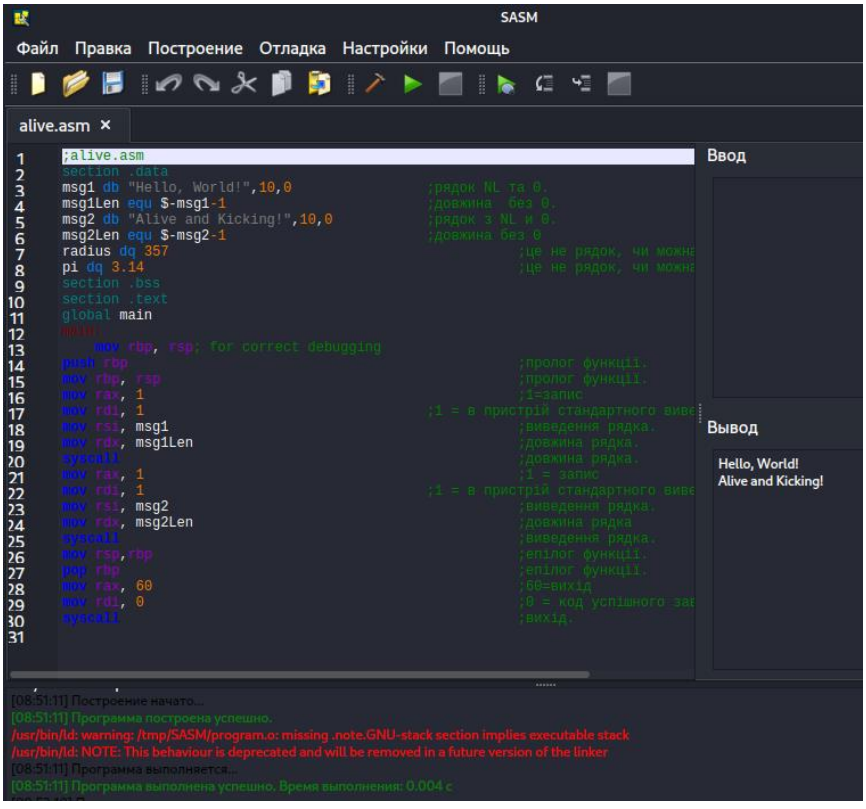



Рисунок 5.16 – Результат виконання програми

У вихідному коді alive.asm використовується дуже зручна функціональна можливість обчислення довжини змінних, як показано нижче:

```
msg1Len equ $-msg1-1
```

Частина інструкції $-msg1-1 означає наступне: взяти цю локацію (адресу) пам’яті ($) і відняти адресу пам’яті, де знаходиться рядок msg1. Результатом є довжина рядка msg1. Ця довжина –1 (віднімається позиція нуля, що завершує рядок) зберігається в константі msg1Len.

Зверніть увагу на використання прологу та епілогу функції в цьому коді. Це необхідно для забезпечення коректної роботи відладника GDB, про що сказано в попередньому розділі. Код прологу та епілогу буде пояснено у наступному розділі.

Виконаємо дослідження оперативної пам’яті з використанням GDB.

```
gdb alive
```

Далі після запрошення (gdb) введіть команду

```
disassemble main
```

Тут видно, що змінна msg1 розташована в пам’яті за адресою 0x4010, це можна перевірити наступною командою (рис 5.17):

```
x/s 0x601030
```

Пара символів \n означає «перехід на новий рядок». Інший спосіб перевірки змінних у GDB (рис 5.17):

```
x/s &msg1
x/dw &radius
x/xw &radius
```

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001130 <+0>:     mov    %rsp,%rbp
   0x0000000000001133 <+3>:     push   %rbp
   0x0000000000001134 <+4>:     mov    %rsp,%rbp
   0x0000000000001137 <+7>:     mov    $0x1,%eax
   0x000000000000113c <+12>:    mov    $0x1,%edi
   0x0000000000001141 <+17>:    movabs $0x4010,%rsi
   0x000000000000114b <+27>:    mov    $0xe,%edx
   0x0000000000001150 <+32>:    syscall
   0x0000000000001152 <+34>:    mov    $0x1,%eax
   0x0000000000001157 <+39>:    mov    $0x1,%edi
   0x000000000000115c <+44>:    movabs $0x401f,%rsi
   0x0000000000001166 <+54>:    mov    $0x13,%edx
   0x000000000000116b <+59>:    syscall
   0x000000000000116d <+61>:    mov    %rbp,%rsp
   0x0000000000001170 <+64>:    pop    %rbp
   0x0000000000001171 <+65>:    mov    $0x3c,%eax
   0x0000000000001176 <+70>:    mov    $0x0,%edi
   0x000000000000117b <+75>:    syscall
   0x000000000000117d <+77>:    nopl   (%rax)
End of assembler dump.
(gdb) x/s 0x4010
0x4010 <msg1>:  "Hello, World!\n"
(gdb) x/s &msg1
0x4010 <msg1>:  "Hello, World!\n"
(gdb) x/dw &radius
0x4033 <radius>:        357
(gdb) x/xw &radius
0x4033 <radius>:        0x00000165
(gdb) x/fg &pi
0x403b <pi>:    3.1400000000000001
(gdb) x/fx &pi
0x403b <pi>:    0x40091eb851eb851f
(gdb)
```

Рисунок 5.17 – Результат відладки програми

Таким чином, отримане десяткове та шістнадцяткове представлення значення, що зберігається в локації пам’яті radius. Для змінних, що містять числа з плаваючою точкою, використовуються такі команди (рис 5.17):

```
x/fg &pi
x/fx &pi
```

Тут існує одна особливість, яку завжди слід пам’ятати. Для наочної демонстрації відкриємо згенерований файл лістингу alive.lst (рис. 5.18).

```
  1                                 ;alive.asm
  2                                 section .data
  3 00000000 48656C6C6F2C20576F-    msg1 db "Hello, World!",10,0          ;рядок NL та 0.
  3 00000009 726C64210A00
  4                                 msg1Len equ $-msg1-1                  ;довжина  без 0.
  5 0000000F 416C69766520616E64-    msg2 db "Alive and Kicking!",10,0     ;рядок з NL и 0.
  5 00000018 204B69636B696E6721-
  5 00000021 0A00
  6                                 msg2Len equ $-msg2-1                  ;довжина без 0
  7 00000023 6501000000000000       radius dq 357                         ;це не рядок, чи можна вивести?
  8 0000002B 1F85EB51B81E0940       pi dq 3.14                                    ;це не рядок, чи можна вивести?
  9                                 section .bss
 10                                 section .text
 11                                 global main
 12                                 main:
 13 00000000 4889E5                     mov rbp, rsp; for correct debugging
 14 00000003 55                     push rbp                                      ;пролог функції.
 15 00000004 4889E5                 mov rbp, rsp                          ;пролог функції.
 16 00000007 B801000000             mov rax, 1                                    ;1=запис
 17 0000000C BF01000000             mov rdi, 1                            ;1 = в пристрій стандартного виведення
dout.
 18 00000011 48BE-                  mov rsi, msg1                         ;виведення рядка.
 18 00000013 [0000000000000000]
 19 0000001B BA0E000000             mov rdx, msg1Len                              ;довжина рядка.
 20 00000020 0F05                   syscall                                       ;довжина рядка.
 21 00000022 B801000000             mov rax, 1                                    ;1 = запис
 22 00000027 BF01000000             mov rdi, 1                            ;1 = в пристрій стандартного виведення
dout.
 23 0000002C 48BE-                  mov rsi, msg2                         ;виведення рядка.
 23 0000002E [0F00000000000000]
 24 00000036 BA13000000             mov rdx, msg2Len                              ;довжина рядка
 25 0000003B 0F05                   syscall                                       ;виведення рядка.
 26 0000003D 4889EC                 mov rsp,rbp                           ;епілог функції.
 27 00000040 5D                     pop rbp                                       ;епілог функції.
 28 00000041 B83C000000             mov rax, 60                                   ;60=вихід
 29 00000046 BF00000000             mov rdi, 0                                    ;0 = код успішного завершення.
 30 0000004B 0F05                   syscall                                       ;вихід.
```

Рисунок 5.18 – Вміст файлу alive.lst

На рядках 7 і 8 ліворуч можна бачити шістнадцяткове представлення значень radius і pi. Замість 0165 записано 6501, а натомість 40091EB851EB851F ми бачимо 1F85EB51B81E0940. Виходить, що байти (1 байт містить два шістнадцяткових числа) зберігаються у зворотному порядку.

Ця особливість називається порядком проходження байтів (endianness). У форматі зі зворотним порядком байтів (від старшого до молодшого – big endian) числа зберігаються у вигляді, у якому ми звикли їх бачити, тобто старші розряди (найбільш значущі цифри) починаються зліва. У форматі з прямим порядком байтів (від молодшого до старшого – little endian) молодші розряди (найменше значущі цифри) починаються зліва. Процесори Intel використовують формат з прямим порядком байтів (little endian), і це може призводити до суттєвих труднощів при читанні коду в шістнадцятковому представленні [29].

**5.3.1 Умовні переходи та цикли в 64-х бітному режимі.** Для дослідження наступних прикладів необхідно встановити IDE SimpleASM (SASM) та виконати наступні налаштування (рис 5.19 – 5.20).

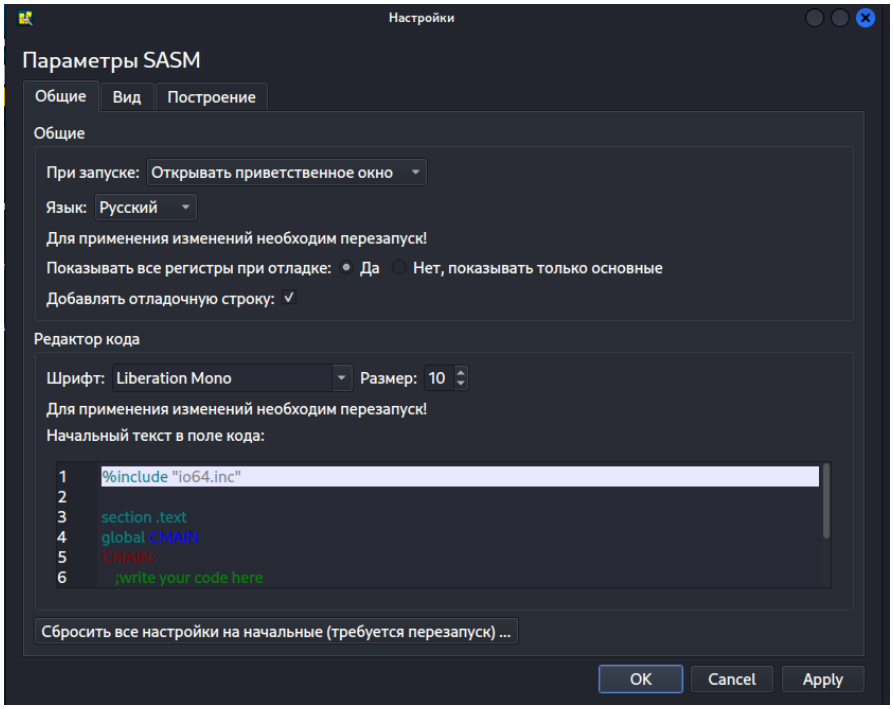



Рисунок 5.19 – Загальні налаштування SASM

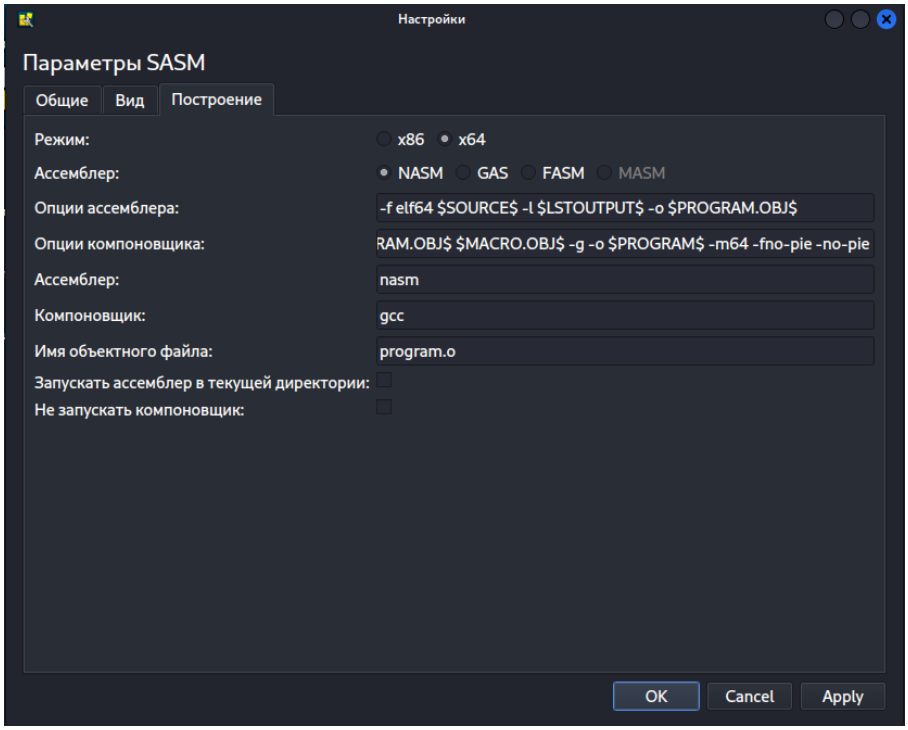



Рисунок 5.20 – Налаштування компіляції та побудови SASM

Лістинг 5.2 – Файл jump.asm

```
;jump.asm
extern printf
section .data
number1 dq 42
number2 dq 41
fmt1 db "NUMBER1 > = NUMBER2",10,0
```

```
fmt2 db "NUMBER1 < NUMBER2",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
mov rax, [number1];Передавання чисел  регістри.
mov rbx, [number2]
cmp rax, rbx      ;Порівняння регістрів rax и rbx.
jge greater       ;Якщо rax більше або дорівнює, то перейти до мітки
                  ;greater:.
mov rdi,fmt2      ;Если rax меньше,продовжити далі.
mov rax,0         ;Регистр xmm не використовується.
call printf       ;Виведення рядка fmt2.
jmp exit          ;Перехід до мітки exit:.
greater:
mov rdi,fmt1      ;Регістр rax більше.
mov rax,0         ;Регістр xmm не не використовується.
call printf       ;Виведення рядка fmt1.
exit:
mov rsp,rbp
pop rbp
ret
```

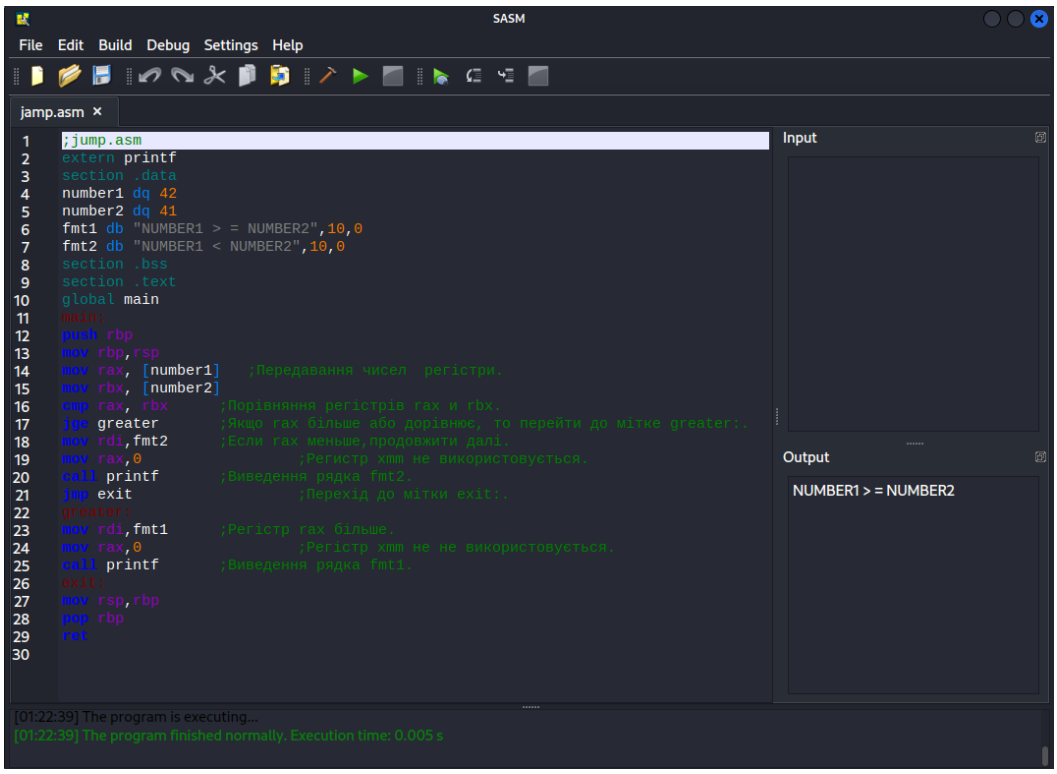



Рисунок 5.21 – Результат виконання програми jamp.asm в SASM

Лістинг 5.3 – Файл makefile для jump.asm

```
#makefile for jamp.asm
jamp: jamp.o
        gcc -o jamp jamp.o -m64 -fno-pie -no-pie
jamp.o: jamp.asm
        nasm -f elf64 -g -F dwarf jamp.asm -l jamp.lst
```

Для початку налагодження програми (рис 5.22) встановіть точки зупинки. У головному меню виберіть Debug (Налагодження) та позначте пункти Show Registers (Показувати регістри) та Show Memory (Показувати пам’ять).

На екрані з’явиться кілька додаткових вікон: Registers (Регістри), Memory (Пам’ять), а також віджет командного рядка GDB.

Тепер за допомогою значка Step (Крок) можна почати покроковий прохід по коду та спостерігати, як змінюються значення в регістрах. Щоб побачити зміни значення змінної, клацніть правою кнопкою миші за оголошенням змінної в розділі section .data і в контекстному меню виберіть пункт Watch (Спостерігати). Ця змінна буде додана у вікно Memory, при цьому SASM спробує автоматично визначити її тип. Якщо значення, виведене SASM, не відповідає очікуваному, необхідно вручну змінити тип на правильний. При налагодженні з використанням SASM для коректного налагодження додається наступний рядок вихідного коду: mov rbp, rsp.

Цей рядок може заплутати інші Debuger-и, такі як GDB, тому не забувайте видаляти його перед запуском GDB окремо з командного рядка.

У головному меню Settings (Параметри налаштування) → Common (Загальні) виберіть пункт Yes (Так) для Show all registers in debug (Показувати всі регістри при налагодженні). При налагодженні у SASM прокрутіть вниз у вікні регістрів. У нижній частині вікна ви побачите 16 ymm регістрів, у кожному з яких вказано два значення у круглих дужках. Перше значення – це відповідний xmm регістр.

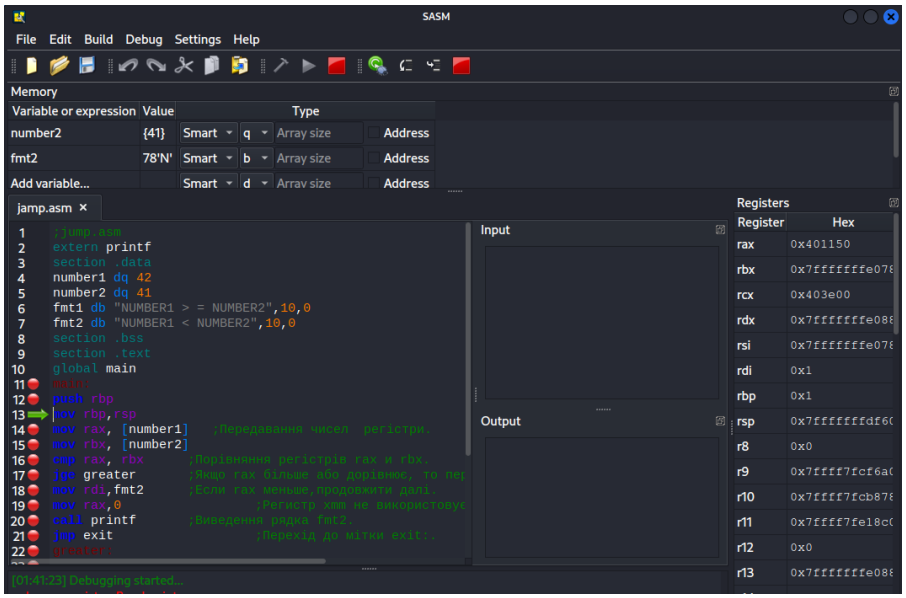



Рисунок 5.22 – Покрокове налагодження програми jamp.asm в SASM

У цій програмі використовувалася інструкція порівняння cmp та дві інструкції переходу jge та jmp. Інструкцію cmp називають інструкцією перевірки умови або просто умовною інструкцією. Тут cmp порівнює два операнди, в даному випадку два регістри. Один з операндів може бути адресою пам’яті, а другий операнд може бути безпосереднім значенням. В будь-якому випадку розмір обох операндів обов’язково має бути однаковим (байт, слово тощо).

Інструкція cmp встановлює чи очищає біти у регістрі прапорів. Докладно інформацію про регістр прапорів та інструкції порівняння, умовних переходів наведено в III частині цього посібника.

Більш складною формою переходу є зациклювання (looping), що означає багаторазове повторення групи інструкцій, поки виконується (або виконується) деяка задана умова. У лістингу 5.4 показаний приклад зациклювання.

Лістинг 5.4 – Програма jumploop.asm

```
;jumploop.asm
extern printf
section .data
number      dq 5
fmt db      "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp, rsp
mov rbx,0         ;Лічильник.
mov rax,0         ;Сума зберігатиметься у регістрі rax.
jloop:
add rax, rbx
inc rbx
cmp rbx,[number]  ;Кінцеве число ітерацій циклу досягнуто?
jle jloop         ;Кінцева кількість ітерацій поки що не досягнута,
                  ;продовження циклу.
                  ; Кінцева кількість ітерацій досягнута, продовжити тут.
mov rdi, fmt; Підготовка результатів.
mov rsi, [number]
mov rdx, rax
mov rax, 0
call printf
mov rsp, rbp
pop rbp
ret
```

Лістинг 5.5 – Файл makefile для jumploop.asm

```
#makefile for jumploop.asm
jumploop: jumploop.o
        gcc -o jumploop jumploop.o -m64 -fno-pie -no-pie
jumploop.o: jumploop.asm
        nasm -f elf64 -g -F dwarf jumploop.asm -l jumploop.lst
```

Ця програма виконує додавання чисел від 0 до значення, що міститься в змінній number. Регістр rbx використовується як лічильник (циклу), а в регістрі rax зберігається сума, що накопичується. Тут створено цикл (loop) – це код між міткою jloop: та інструкцією jle jloop. У цьому циклі значення регістру rbx додається до значення регістра rax, після чого значення регістра rbx збільшується на 1, потім виконується порівняння, щоб дізнатися, чи не було досягнуто кінцевого значення (number). Якщо регістрі rbx міститься значення, менше чи рівне значенням number, то виконання циклу триває, інакше виконання триває з інструкції, наступної безпосередньо після циклу, і починається підготовка до

висновку результату. Для збільшення значення регістру rbx використано арифметична інструкція inc. У лістингу 5.6 показаний інший спосіб створення циклу.

Лістинг 5.6 – Програма betterloop.asm

```
; betterloop
extern printf
section .data
number dq 5
fmt db "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
mov rcx, [number]       ;Ініціалізація регістра rcx значенням number.
mov rax, 0
bloop:
add rax,rcx             ; Додавання rcx для отримання суми.
loop bloop              ; Цикл, доки значення rcx зменшується на 1.
                        ; До того часу, коли виконається умова rcx = 0.
mov rdi,fmt             ; rcx = 0, продовжити тут.
mov rsi, [number]       ; Сума, що виводиться.
mov rdx, rax
mov rax,0               ; Без використання чисел із плаваючою точкою.
call printf             ; Виведення результату.
mov rsp,rbp
pop rbp
ret
```

Тут можна бачити, що існує спеціалізована інструкція, яка використовує регістр rcx як лічильник циклу, що зменшується. При кожному проході по циклу значення rcx автоматично зменшується на 1, і показ rcx не дорівнює 0, цикл виконується знову і знову. У цьому варіанті обсяг вихідного коду, що вводиться менше. Докладно інформацію про цикли наведено в III частині цього посібника [29].

**5.3.2 Основи цілочислової арифметики в 64-х бітному режимі.** В лістингу 5.7 показано приклад коду, який демонструє арифметичні операції [29].

Лістинг 5.7 – Програма icalc.asm

```
;icalc.asm
extern printf
section .data
number1 dq 128          ;Числа, які використовуються для демонстрації
number2 dq 19           ;арифметичних обчислень
neg_num dq -12          ;та розповсюдження знакового розряду.
fmt db "The numbers are %ld and %ld",10,0
fmtint db "%s %ld",10,0
sumi db "The sum is",0
difi db "The difference is",0
inci db "Number 1 Incremented:",0
deci db "Number 1 Decremented:",0
sali db "Number 1 Shift left 2 (x4):",0
sari db "Number 1 Shift right 2 (/4):",0
```

```
sariex db "Number 1 Shift right 2 (/4) with "
db "sign extension:",0
multi db "The product is",0
divi db "The integer quotient is",0
remi db "The modulo is",0
section .bss
resulti resq 1
modulo resq 1
section .text
global main
main:
push rbp
mov rbp,rsp
; Виведення чисел.
mov rdi, fmt
mov rsi, [number1]
mov rdx, [number2]
mov rax, 0
call printf
; Додавання ----------------------------------------------------------
mov rax, [number1]
add rax, [number2]      ; Додавання number2 с rax.
mov [resulti], rax      ; Переміщення суми в змінну result.
; Виведення результату
mov rdi, fmtint
mov rsi, sumi
mov rdx, [resulti]
mov rax, 0
call printf
; Віднімання----------------------------------------------------------
mov rax, [number1]
sub rax, [number2]      ; Віднімання number2 з rax.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, difi
mov rdx, [resulti]
mov rax, 0
call printf
; Інкркмент----------------------------------------------------------
mov rax, [number1]
inc rax                 ; Інкремент rax на 1.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, inci
mov rdx, [resulti]
mov rax, 0
call printf
; Декремент----------------------------------------------------------
mov rax, [number1]
dec rax           ; Декрекмент rax на 1.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, deci
mov rdx, [resulti]
mov rax, 0
call printf
; Арифметичний зсув ліворуч-----------------------------------------
mov rax, [number1]
sal rax, 2        ; Множення rax на 4.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
```

```
mov rsi, sali
mov rdx, [resulti]
mov rax, 0
call printf
; Арифметичний зсув праворуч----------------------------------------
mov rax, [number1]
sar rax, 2        ; Ділення rax на 4.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, sari
mov rdx, [resulti]
mov rax, 0
call printf
; Арифметичний зсув праворуч з розповсюдженням знакового розряду -----
mov rax, [neg_num]
sar rax, 2        ; Ділення rax на 4.
mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, sariex
mov rdx, [resulti]
mov rax, 0
call printf
; Множення-----------------------------------------------------------
mov rax, [number1]
imul qword [number2]    ; Множення rax на number2.
 mov [resulti], rax
; Виведення результату
mov rdi, fmtint
mov rsi, multi
mov rdx, [resulti]
mov rax, 0
call printf
; Деление------------------------------------------------------------
mov rax, [number1]
mov rdx, 0              ; В rdx повинен бути 0 перед idiv.
idiv qword [number2]    ; Ділення rax на number2, залишок в rdx.
mov [resulti], rax
mov [modulo], rdx       ; Запис вмісту rdx в modulo.
; Виведення результату
mov rdi, fmtint
mov rsi, divi
mov rdx, [resulti]
mov rax, 0
call printf
mov rdi, fmtint
mov rsi, remi
mov rdx, [modulo]
mov rax, 0
call printf
mov rsp,rbp
pop rbp
ret
```

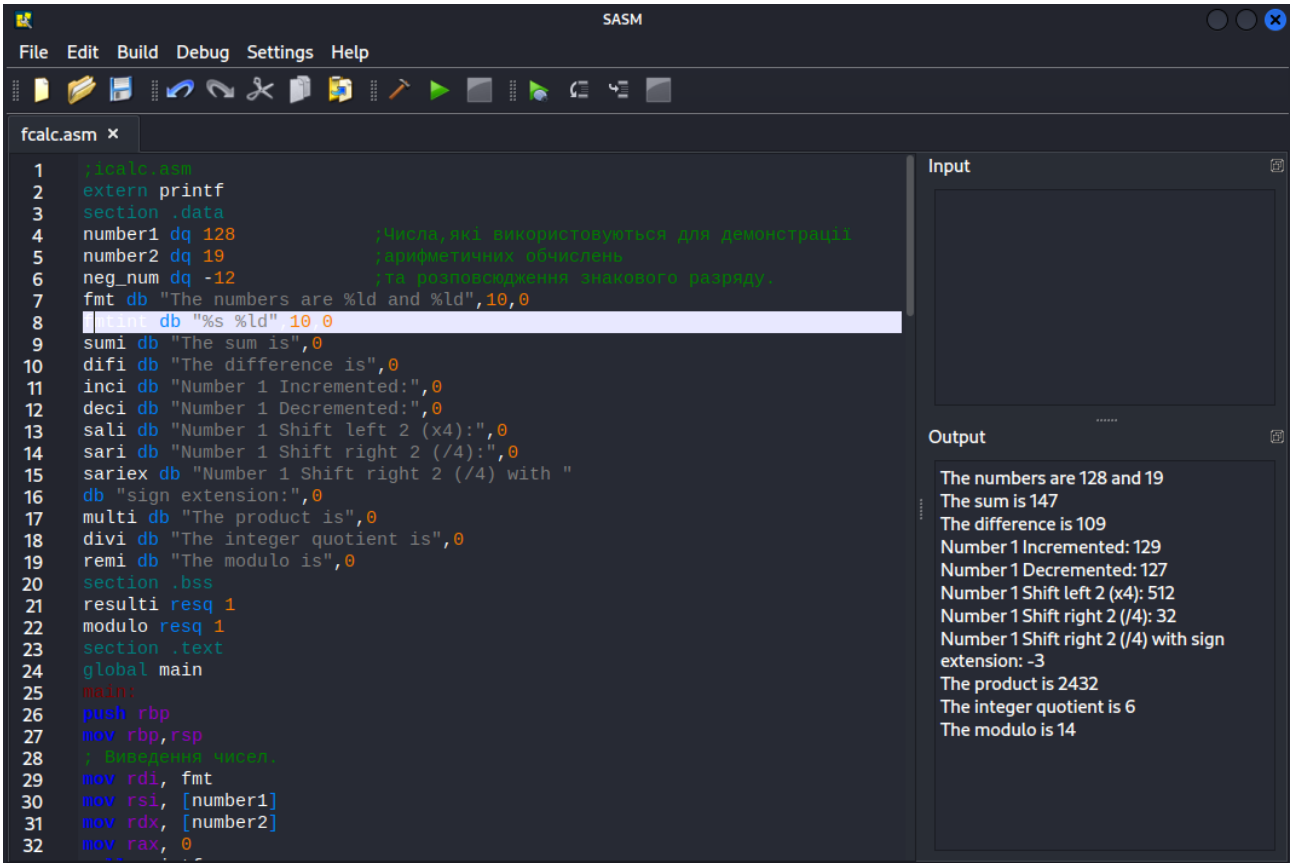



Рисунок 5.23 – Результат виконання програми icalc.asm

Докладно інформацію про арифметичні операції наведено в III частині цього посібника.

**5.3.3 Основи арифметики з плаваючою комою в 64-х бітному режимі.** Число зі звичайною (одиничною) точністю (single precision) зберігається в 32 бітах: 1 знаковий біт, 8 біт показника степеню та 23 біта дробової частини.

| S | EEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFF |
|---|---|---|
| 0 | 1 8 | 9 31 |

Число з подвійною точністю (double precision) зберігається в 64 бітах: 1 знаковий біт, 11 біт показника степеню та 52 біти дробової частини.

| S | EEEEEEEEEEE | FFFFFFFFFFF…FFFFFFFFFFFF |
|---|---|---|
| 0 | 1 11 | 12 63 |

Знаковий біт пояснюється просто: якщо число додатне, цей біт дорівнює 0, якщо число від’ємне, цей біт дорівнює 1.

Сенс бітів показника степеню пояснити трохи складніше. Розглянемо приклад із десятковими числами.

$$200 = 2{,}0 \times 102$$
$$5000{,}30 = 5{,}0003 \times 103$$

Тепер приклад із двійковими числами:

$1101010.01011 = 1.0101001011 \times 2^6$ (дробова точка переміщена на 6 позицій вліво)

Але показник степеню може бути додатним, від'ємним або дорівнювати нулю. Щоб зробити ці відмінності зрозумілими, для чисел зі звичайною точністю до додатного показника степеню додається число 127 перед збереженням. Це означає, що нульовий показник степеню буде збережено як 127. Це число 127 називається зміщенням (bias). Для чисел з подвійною точністю зміщення дорівнює 1023.

У наведеному вище прикладі число 1.0101001011 називається мантисою (significand; mantissa). Передбачається, що перший біт мантиси завжди дорівнює 1 (тобто мантиса «нормалізована»), тому він не зберігається. Нижче наводиться простий приклад, який показує, як це працює.

Приклад з десятковим числом із звичайною точністю 10:

- десяткове число 10 має вигляд 1010 як двійкове ціле число;
- знаковий біт дорівнює 0, тому що число додатне;
- отримано число у форматі b.bbbb. Тут 1.010 – мантиса з першої 1 відповідно до вимог. Найперша 1 не буде збережена;
- отже, показник степеню дорівнює 3, тому що дробова точка була переміщена на три позиції вліво. Показник степеню додатній, до нього додається 127, в результаті показник степеню дорівнює 130 або в двійковому форматі 10000010;
- таким чином, десяткове число зі звичайною точністю 10 буде збережено в наступному вигляді:

| | | |
|---|---|---|
| 0 | 10000010 | 01000000000000000000000 |
| S | EEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFF |

або 41200000 у шістнадцятковому форматі.

Слід зазначити, що шістнадцяткове уявлення значення звичайної точності відрізняється від того ж значення, поданого з подвійною точністю. Обчислення з подвійною точністю повільніше обчислень зі звичайною точністю, а операнди з подвійною точністю вимагають більше пам'яті.

У старіших програмах можуть зустрітися 80 бітові числа з плаваючою точкою, але для таких чисел існують власні спеціалізовані інструкції, що позначаються як FPU інструкції. Ця функціональність є спадщиною минулого і не рекомендується до використання.

Порівняння чисел зі звичайною та подвійною точністю у нових розробках. Однак у статтях в інтернеті час від часу виявляються FPU інструкції.

У лістингу 5.8 наведено приклад програми, яка використовує числа з плаваючою точкою.

Лістинг 5.8 – Програма fcalc.asm

```
; fcalc.asm
extern printf
section .data
number1     dq 9.0
number2     dq 73.0
```

```
fmt         db "Numbers are %f and %f",10,0
fmt float   db "%s %f",10,0
f_sum       db "Це float sum of %f and %f is %f",10,0
f_dif       db "The float difference of %f and %f is %f",10,0
f_mul       db "Фруктовий продукт %f і %f є %f",10,0
f_div       db "The float division of %f by %f is %f",10,0
f_sqrt      db "The float squareroot of %f is %f",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
; Виведення чисел.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, fmt
mov rax, 2              ; Два числа з плаваючою точкою.
call printf
; Обчислення суми.
movsd xmm2, [number1]   ; Число з плаваючою точкою подвійної точності xmm.
addsd xmm2, [number2]   ; Додавання з іншим числом подвійної точності,
                        ;результат xmm.
; Виведення результату.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, f_sum
mov rax, 3              ; Три числа з плаваючою точкою.
call printf
; Обчислення різниці.
movsd xmm2, [number1]   ; Число з плаваючою точкою подвійної точності xmm.
subsd xmm2, [number2]   ; Віднімання іншого числа подвійної точності з xmm.
; Виведення результату.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, f_dif
mov rax, 3              ; Три числа з плаваючою точкою.
call printf

; множення.
movsd xmm2, [number1]   ; Число з плаваючою точкою подвійної точності xmm.
mulsd xmm2, [number2]   ; Збільшення заданого числа на xmm.
; Виведення результату.
mov rdi,f_mul
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,3 ; Три числа з плаваючою точкою.
call printf
; Ділення.
movsd xmm2, [number1] ; Число з плаваючою точкою подвійної точності xmm.
divsd xmm2, [number2] ; Ділення xmm0.
; Виведення результату.
mov rdi,f_div
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,1               ; Одне число з плаваючою точкою.
call printf
; Обчислення квадратного кореня.
sqrtsd xmm1, [number1] ; Квадратний корінь подвійної точності xmm.
; Виведення результату.
mov rdi, f_sqrt
movsd xmm0, [number1]
mov rax, 2              ; Два числа з плаваючою точкою.
call printf
; Вихід з програми.
```

```
mov rsp, rbp
pop rbp             ; Операція, обернена операції push на початку програми.
ret
```

Це проста програма – насправді організація виведення вимагає більше зусиль, ніж обчислення чисел з плаваючою комою.

Арифметичні інструкції для чисел із звичайною точністю: addss, subss, mulss, divss та sqrtss [29].

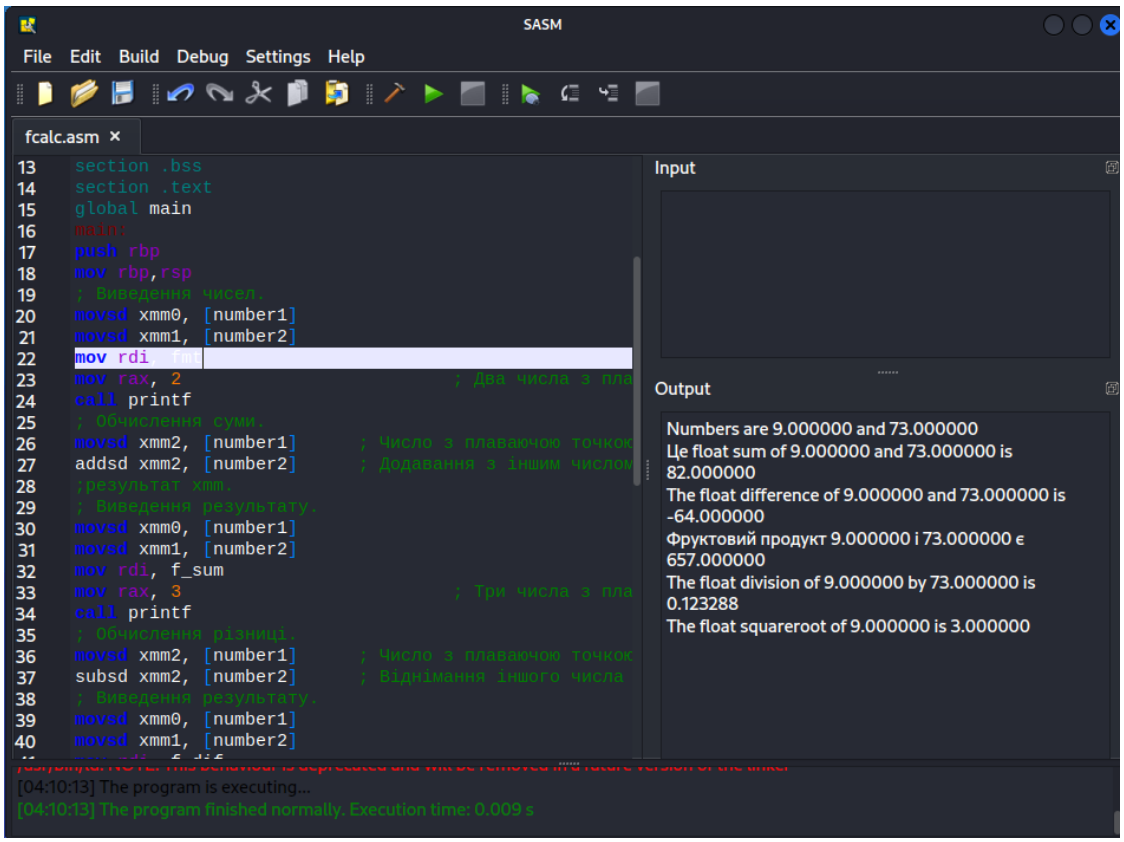



Рисунок 5.24 – Результат виконання програми pcalc.asm

**5.3.4 Функції.** Асемблер не є «структурною мовою» бо містить численні інструкції jmp та мітки, які дозволяють під час виконання програми переходити та повертатися назад. У сучасних мовах програмування високого рівня існують такі програмні конструкції, як do...while, while...do, case і т. п. Подібних конструкцій немає в мові асемблера.

Але, як і в сучасних мовах програмування, в мові асемблера є функції та процедури, які допомагають зробити вихідний код структурованим. Невелике уточнення: функція виконує інструкції та повертає деяке значення. Процедура виконує вказівки, але не повертає значення.

У лістингу 5.9 показаний приклад програми на асемблері із простою функцією, що обчислює площу кола.

Лістинг 5.9 – Програма function.asm

```
; function.asm
extern printf
section .data
```

```
radius      dq 10.0
pi          dq 3.14
fmt         db "Area of the circle is %.2f",10,0
section .bss
section .text
global main
;-------------------------------------------
main:
push rbp
mov rbp, rsp
call area               ; Виклик функції.
mov rdi,fmt
; Формат виведення.
movsd xmm1, [radius]    ; Запис числа з плавною точкою в регістр xmm1.
mov rax,1
; Значення площі у регістрі xmm0.
call printf
leave
ret
;-------------------------------------------
area:
push rbp
mov rbp, rsp
movsd xmm0, [radius]    ; Запис числа з плавною точкою в регістр xmm0.
mulsd xmm0, [radius]    ; Множення xmm0 на число з плавною точкою.
mulsd xmm0, [pi]        ; Множення xmm0 на (інше) число з плавною точкою.
leave
ret
```

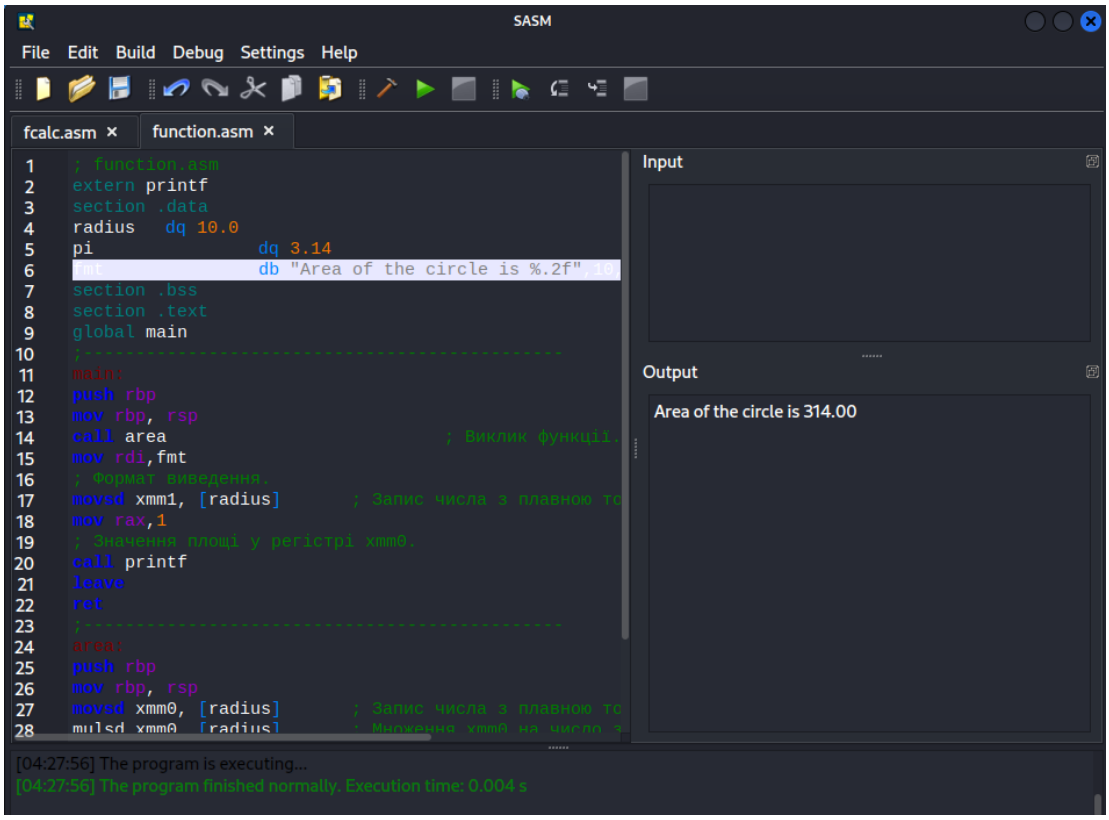



Рисунок 5.25 – Результат виконання програми function.asm

У цій програмі є основна частина, яка визначається, як і раніше, міткою main, а за нею слідує функція, яка визначається міткою area. В основному main викликається функція area, що обчислює площу круга з використанням radius та

pi – змінних, значення яких зберігаються у блоках пам’яті. Тут можна бачити, що функції обов’язково повинні мати пролог та епілог, так само, як і основна частина main.

Обчислена площа круга зберігається у регістрі xmm0. Після повернення з функції в основну частину main викликається функція printf з регістром rax, що містить 1, це означає, що один xmm регістр повинен бути виведений. Тут вводиться нова інструкція leave, яка виконує ті ж дії, що й mov rsp, rbp та pop rbp (епілог) [29].

**5.3.5 Зовнішні функції.** Функції не обов’язково повинні розміщуватись у тому ж файлі, в якому знаходиться основна програма. Можна написати та ассемблювати функції в окремому файлі та пов’язувати їх під час складання програми. Функція printf, є прикладом зовнішньої функції. У файлі вихідного коду, де планується застосування зовнішньої функції необхідно оголосити її за допомогою ключового слова extern, щоб повідомити асемблеру про те, що вихідний код цієї функції не слідує шукати у поточному файлі. Асемблер припускає, що ця функція вже ассембльована в деякому об’єктному файлі. Зовнішню функцію буде вставлено лінкером, якщо він зможе знайти її у заданому об’єктному файлі. Можна використовувати зовнішні функції з мови С, але також можна створити власний набір зовнішніх функцій і за потреби пов’язувати їх із основною програмою.

У лістингу 5.10 показаний приклад програми, що складається з трьох файлів вихідного коду з іменами: function4.asm, circle.asm та rect.asm. Для цієї програми написано новий makefile. Уважно вивчіть вміст цих файлів.

Лістинг 5.10 – Програма function4.asm

```
; function4.asm
extern printf
extern c_area
extern c_circum
extern r_area
extern r_circum
global pi
section .data
pi          dq 3.141592654
radius      dq 10.0
side1       dq 4
side2       dq 5
fmtf        db "%s %f",10,0
fmti        db "%s%d",10,0
ca          db "The circle area is ",0
cc          db "The circle circumference is ",0
ra          db "The rectangle area is ",0
rc          db "The rectangle circumference is ",0
section .bss
section .text
global main
main:
push rbp
mov rbp, rsp
; Площа круга.
movsd xmm0, qword [radius]    ; Аргумент radius у регістрі xmm0.
call c_area
```

```
; Площа повертається у регістрі xmm0.
; Виведення площі круга.
mov rdi, fmtf
mov rsi, ca
mov rax, 1
call printf
; Довжина кола.
movsd
xmm0, qword [radius]          ; Аргумент radius у регістрі xmm0.
call c_circum
; Довжина кола в регістрі xmm0.
; Виведення довжини кола.
mov rdi, fmtf
mov rsi, cc
mov rax, 1
call printf
; Площа прямокутника.
mov rdi, [side1]
mov rsi, [side2]
call r_area
; Площа повертається у регістрі rax.
; Виведення площі прямокутника.
mov rdi, fmti
mov rsi, ra
mov rdx, rax
mov rax, 0
call printf
; Периметр прямокутника.
mov rdi, [side1]
mov rsi, [side2]
call r_circum
; Периметр у регістрі rax.
; Виведення периметра прямокутника.
mov rdi, fmti
mov rsi, rc
mov rdx, rax
mov rax, 0
call printf
mov rsp,rbp
pop rbp
ret
```

У вихідному коді у лістингу 5.10 кілька функцій оголошено як extern (зовнішні), як це раніше робилося неодноразово при використанні функції printf. Крім того, змінна pi оголошена як Global. Таким чином, ця змінна буде доступною у зовнішніх функціях. У лістингах 5.11 та 5.12 показані окремі файли, що містять тільки зовнішні функції.

Лістинг 5.11 - Файл circle.asm

```
; circle.asm
extern pi
section .data
section .bss
section .text
;-----------------------------------------------
global c_area
c_area:
section .text
push rbp
mov rbp,rsp
movsd xmm1, qword [pi]
mulsd xmm0, xmm0        ; Радіус у регістрі xmm0.
mulsd xmm0, xmm1
```

```
mov rsp,rbp
pop rbp
ret
;-----------------------------------------------
global c_circum
c_circum:
section .text
push rbp
mov rbp,rsp
movsd xmm1, qword [pi]
addsd xmm0, xmm0        ; Радіус у регістрі xmm0.
mulsd xmm0, xmm1
mov rsp,rbp
pop rbp
ret
```

Лістинг 5.12 – Файл rect.asm

```
; rect.asm
section .data
section .bss
section .text
;-----------------------------------------------
global r_area
r_area:
section .text
push rbp
mov rbp,rsp
mov rax, rsi
imul rax, rdi
mov rsp,rbp
pop rbp
ret
;-----------------------------------------------
global r_circum
r_circum:
section .text
push rbp
mov rbp,rsp
mov rax, rsi
add rax, rdi
add rax, rax
mov rsp,rbp
pop rbp
ret
```

У файлі circle.asm необхідно використовувати змінну pi, оголошену в основному файлі вихідного коду як global, що необхідно відзначити як не найвдаліше рішення, але тут воно застосовано з демонстраційною метою. Глобальні змінні, подібні до pi, важко відстежувати, і вони можуть навіть призводити до конфлікту змінних із однаковими іменами. Саме найкраще практичне рішення – використання регістрів передачі значень у функцію. Тут доводиться визначати pi як зовнішню змінну. Кожен файл circle.asm та rect.asm містить дві функції: для обчислення периметра та площі. Необхідно оголосити, що це функції є глобальними (global), як і основна функція main. Під час ассемблювання цих функцій додаються необхідні «службові дані», що дозволяють лінкеру включати ці функції до іншого об’єктного коду.

Для виконання всієї роботи потрібна розширена версія makefile, показана в лістингу 5.13

Лістинг 5.13 – makefile

```
# makefile for function4, circle and rect.
function4: function4.o circle.o rect.o
gcc -g -o function4 function4.o circle.o rect.o -no -pie
function4.o: function4.asm
nasm -f elf64 -g -F dwarf function4.asm -l function4.lst
circle.o: circle.asm
nasm -f elf64 -g -F dwarf circle.asm -l circle.lst
rect.o: rect.asm
nasm -f elf64 -g -F dwarf rect.asm -l rect.lst
```

Читати makefile потрібно знизу вгору: спочатку різні вихідні файли коди на асемблері ассемблюються в об'єктні файли, потім ці об'єктні файли зв'язуються і об'єднуються у виконуваний файл function4. При будь-якій зміні одного з файлів вихідного коду make завдяки деревоподібній структурі точно знає, які файли потрібно повторно ассемблювати та зв'язати. Зрозуміло, якщо функції стабільні і надалі не будуть змінюватися, то немає необхідності їх повторного ассемблювання при кожному сеансі обробки makefile. Просто збережіть об'єктні файли у спеціально створеному каталозі та надалі звертайтеся до них за повним ім'ям у рядку gcc у makefile. Об'єктний файл – результат ассемблювання або компіляції вихідного коду. Об'єктний файл містить машинний код та інформацію для лінкера про глобальні змінні та зовнішні функції, необхідні для створення коректного виконуваного файлу. У цьому прикладі всі об'єктні файли розміщені в тому ж каталозі, що і основний файл вихідного коду, тому повні дорожні імена не використовуються.

Компілятор gcc має достатній інтелект, для того що б пошукати у бібліотеках мови С функції, звернення до яких зустрічаються у вихідному коді. Це означає, що не потрібно використовувати імена функцій мови С для назви своїх функцій. Це може заплутати будь-кого, не кажучи вже про лінкер.

У наведеному вище коді регістри використовувалися для передачі значень з основної програми у функції та у зворотному напрямку – це найбільш правильний практичний підхід. Наприклад, перед викликом r_area значення змінної side1 записується в регістр rdi, а значення side2 у регістр rsi.

Потім обчислена площа прямокутника повертається регістрі rax. Для повернення результату можна було б скористатися глобальною змінною, подібною до pi, оголошеної в розділі section .data функції main. Але, як вже було зазначено вище, цього слід уникати [29].

**5.3.6 Угоди про виклики функцій.** Ці угоди описують, як змінні передаються у функції та повертаються з функцій. Якщо використовувати функції з бібліотеки мови С, необхідно знати, в які регістри потрібно записувати значення, з якими мають працювати такі зовнішні функції. Крім того, якщо пишуться асемблерні функції для створення бібліотеки, призначеної для використання іншими розробниками, то рекомендується дотримуватись деяких угод про регістри, в яких передаються конкретні аргументи функцій. Без дотримання цих угод можуть постійно виникати проблеми та конфлікти з передачею аргументів.

Щоб уникнути конфліктів та їх наслідків у вигляді критичних збоїв програми, досвідчені програмісти розробили угоди про виклики функцій (Calling conventions), стандартизований спосіб виклику функцій. Тут використовуються угоди про виклик функцій System V AMD64 ABI, які є стандартом платформ Linux.

Але існують також і інші угоди про виклик функцій, що варті уваги: Microsoft x64, що використовуються для програмування в ОС Windows.

Ці угоди про виклики функцій дозволяють використовувати зовнішні функції, написані на асемблері, а також функції, скомпільовані з вихідного коду іншими мовами, такими як C, без необхідності доступу до їх вихідного коду. Потрібно просто записати правильні аргументи у регістри, визначені в угоді про виклик функції.

Більш детальну інформацію про угоди про функцій System V AMD64 ABI можна знайти тут: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf. Цей документ компанії Intel містить величезний обсяг детальної інформації про прикладний двійковий (бінарний) інтерфейс (ABI) System V.

**Аргументи функцій.** Аргументи, які не є значеннями з плаваючою точкою, такі як цілі числа та адреси, передаються в наступному порядку:

- 1-й аргумент передається в регістрі rdi;
- 2-й аргумент передається в регістрі rsi;
- 3-й аргумент передається в регістрі rdx;
- 4-й аргумент передається в регістрі rcx;
- 5-й аргумент передається в регістрі r8;
- 6-й аргумент передається в регістрі r9.

Додаткові аргументи передаються через стек – обов’язково у зворотному порядку, щоб можна було витягти їх у правильному порядку. Наприклад, при передачі 10 аргументів порядок наступний:

- першим у стек записується 10-й аргумент;
- потім у стек записується 9-й аргумент;
- потім у стек записується 8-й аргумент;
- останнім у стек записується 7-й аргумент.

Після переходу у функцію необхідно забезпечити вилучення значень із регістрів. При вилученні значень зі стека слід пам’ятати про те, що адреса повернення також записується в стек відразу після аргументів:

- при записі в стек 10-го аргументу покажчик стека rsp зменшується на 8 байт;
- при записі в стек 9-го аргументу rsp зменшується на 8 байт;
- при записі в стек 8-го аргументу rsp зменшується на 8 байт;
- при записі в стек 7-го аргументу rsp зменшується на 8 байт;
- потім при виклику функції вміст регістру rip записується в стек і покажчик стека rsp зменшується на 8 байт;
- далі на початку функції регістр rbp записується в стек – це частина прологу, а покажчик стека rsp зменшується на 8 байт;

- після цього стек вирівнюється по 16-ти байтовій межі, тому, можливо, знадобиться ще одна операція запису в стек push, щоб зменшити RSP на 8 байт.

Таким чином, після запису в стек аргументів функції, як мінімум, ще два додаткові регістри записуються в стек, тобто 16 додаткових байт. Тому після входу в функцію доступу до аргументів необхідно пропустити перші 16 байт у стеку або, можливо, більше, якщо виконувалась операція вирівнювання стека.

Аргументи з плаваючою точкою передаються в xmm-регістрах в наступному порядку:

- 1-й аргумент записується в регістр xmm0;
- 2-й аргумент записується в регістр xmm1;
- 3-й аргумент записується в регістр xmm2;
- 4-й аргумент записується в регістр xmm3;
- 5-й аргумент записується в регістр xmm4;
- 6-й аргумент записується в регістр xmm5;
- 7-й аргумент записується в регістр xmm6;
- 8-й аргумент записується в регістр xmm7.

Додаткові аргументи передаються через стек, але запис у стек виконується за допомогою інструкції push, як можна було припустити.

Функція повертає результат з плаваючою точкою в регістрі xmm0, а ціле число або адреса повертається в регістрі rax.

У лістингу 5.14 показаний приклад, який виводить кілька аргументів за допомогою функції printf.

Лістинг 5.14 – Програма function5.asm

```
; function5.asm
extern printf
section .data
first       db "A",0
second      db "B",0
third       db "C",0
fourth      db "D",0
fifth       db "E",0
sixth       db "F",0
seventh     db "G",0
eighth      db "H",0
ninth       db "I",0
tenth       db "J",0
fmt1        db "The string is: %s%s%s%s%s%s%s%s%s%s",10,0
fmt2        db "PI = %f",10,0
pi          dq 3.14
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
mov rdi, fmt1                ; Спочатку використовуються регістри.
mov rsi, first
mov rdx, second
mov rcx, third
mov r8, fourth
mov r9, fifth
```

```
push tenth                      ; Тепер починається запис у стек
push ninth                      ; в зворотному напрямку.
push eighth
push seventh
push sixth
mov rax, 0
call printf
and rsp, 0xfffffffffffffff0; Вирівнювання стеку по 16-байтовій межі.
Movsd xmm0, [pi]                ; Виведення числа з плаваючою точкою.
mov rax, 1                      ; Виводиться 1 число з плавною точкою.
mov rdi, fmt2
call printf
leave
ret
```

У цьому прикладі всі аргументи передаються у правильному порядку у функцію printf. Зверніть увагу: у стек аргументи записуються у зворотному порядку.

Перед викликом функції printf використовуйте debugger для перевірки вмісту регістру rsp. Стек не вирівняний по 16 байтовій межі. В програмі не виникає критичний збій, тому що від printf не вимагається виведення числа з плаваючою точкою. Але за наступного виклику printf виконується саме таке виведення. Тому перед використанням printf необхідно вирівняти стек, для цього використовується інструкція and rsp, 0xfffffffffffffff0

Ця інструкція зберігає всі байти в регістрі rsp без змін, за винятком останнього байта: останні (молодші) 4 біти в rsp змінюються на 0, тобто числове значення rsp зменшується, і покажчик стека rsp вирівнюється по 16 байтовій межі. Якщо стек вже був вирівняний спочатку, то наведена вище інструкція нічого не робить. Проте слід бути уважним та обережним. Якщо необхідно витягувати значення зі стека після виконання цієї інструкції and, виникає проблема: необхідно дізнатися, чи змінила інструкція and значення rsp, і в результаті вирівняти покажчик стека знову до цього значення перед виконанням інструкції and.

**Схема стеку.** Розглянемо приклад, у якому можна спостерігати, що відбувається у стеку під час запису до нього аргументів функції. У лістингу 5.15 показана програма, що використовується функцією для формування рядка, а після повернення з функції створений рядок виводиться.

Лістинг 5.15 - Програма function6.asm

```
; function6.asm
extern printf
section .data
first       db "A"
second      db "B"
third       db "C"
fourth      db "D"
fifth       db "E"
sixth       db "F"
seventh     db "G"
eighth      db "H"
ninth       db "I"
tenth       db "J"
fmt         db "The string is: %s",10,0
```

```
section .bss
flist       resb 11; Довжина рядка + завершальний 0.
section .text
global main
main:
push rbp
mov rbp, rsp
mov rdi, flist          ; довжина.
mov rsi, first          ; Заповнення регістрів.
mov rdx, second
mov rcx, third
mov r8, fourth
mov r9, fifth
push tenth              ; Тепер починається запис у стек
push ninth              ; в зворотному напрямку.
push eighth
push seventh
push sixth
call lfunc              ; Виклик функції.
; Виведення результату.
mov rdi, fmt
mov rsi, flist
mov rax, 0
call printf
leave
ret
;-----------------------------------------------
lfunc:
push rbp
mov rbp,rsp
xor rax,rax             ; Очищення rax (особливо старші біти).
mov al, byte [rsi]      ; Запис значення 1-го аргументу al.
mov [rdi], al           ; Збереження al у пам’яті.
mov al, byte [rdx]      ; Запис значення 2-го аргументу al.
mov [rdi+1], al         ; Збереження al у пам’яті.
mov al, byte[rcx]       ; І т.д . для інших аргументів.
mov [rdi+2], al
mov al, byte[r8]
mov [rdi+3], al
mov al, byte[r9]
mov [rdi+4], al
; Тепер вилучення аргументів зі стеку.
push rbx    ; Зберігається функцією, що викликається.
xor rbx,rbx
mov rax, qword [rbp+16]; Перше значення: початковий покажчик стека
                       ;+rip+rbp
mov bl, byte[rax]       ; Вилучення символу.
mov [rdi+5], bl         ; Збереження символу у пам’яті.
mov rax, qword [rbp+24] ; Продовження обробки наступного значення.
mov bl, byte[rax]
mov [rdi+6], bl
mov rax, qword [rbp+32]
mov bl, byte[rax]
mov [rdi+7], bl
mov rax, qword [rbp+40]
mov bl, byte[rax]
mov [rdi+8], bl
mov rax, qword [rbp+48]
mov bl, byte[rax]
mov [rdi+9], bl
mov bl,0
mov [rdi+10], bl
pop rbx ; Зберігається функцією, що викликається.
mov rsp,rbp
pop rbp
```

```
ret
```

У цьому прикладі замість виведення за допомогою функції printf відразу після передачі всіх аргументів, викликається функція lfunc. Ця функція приймає всі аргументи та формує рядок у пам’яті (flist). Цей рядок буде виведено після повернення до основної функції main.

Розглянемо докладніше функцію lfunc. Вона приймає лише молодший байт регістрів аргументів, у яких записані символи, використовуючи інструкцію mov al, byte[rsi].

Ці символи по черзі зберігаються у пам’яті, починаючи з адреси, що зберігається в регістрі rdi, тобто з адреси flist, інструкцією mov [rdi], al. Ізолювання ключового слова byte не обов’язкове, але воно підвищує зручність читання вихідного коду.

На початку функції lfunc значення rsp, тобто адреса стека, зберігається в регістрі rbp. Але між цією інструкцією та кінцевою інструкцією запису значень у стек у main вміст регістру rsp було змінено двічі. По-перше, при виклику lfunc адреса повернення була поміщена в стек. Потім у стеку було збережено регістр rbp як частину прологу. У сумі покажчик стека RSP зменшився на 16 байт. Для доступу до значень аргументів у стеку необхідно скоригувати їх адреси на 16 байт. Саме тому для доступу, наприклад, до змінної sixth використовується інструкція mov rax, qword [rbp+16].

Кожна наступна змінна розташовується на 8 байт вище за попередню. Регістр rbx використовується як тимчасове сховище для формування рядка в пам’яті flist. Перед застосуванням rbx його вміст зберігається у стеку. Ніколи не відомо, чи застосовується регістр rbx в основній програмі main для інших цілей, тому його вміст зберігається та відновлюється перед виходом із функції.

**Збереження регістрів.** Тепер необхідно пояснити зміст інструкцій:

```
push  rbx ; Зберігається функцією, що викликається.
i
pop rbx ; Зберігається функцією, що викликається.
```

Має бути очевидною необхідність спостереження за тим, що відбувається з регістром під час виклику функції. Деякі регістри змінюватимуться під час виконання функції інші залишаться незмінними. Необхідно вжити запобіжних заходів, щоб уникнути непередбачуваних результатів через зміни регістрів функціями, що використовуються в основній (що викликає) програмі main.

У табл. 5.3 надано короткий огляд визначень призначення регістрів з угод про виклики функцій.

Функція, що викликається в документації, позначена як callee. Якщо функція використовує регістр, що зберігається функцією, яка викликається (calee-saved register), то вона обов’язково повинна помістити вміст цього регістру в стек перед його використанням та перед завершенням роботи відновити вміст у правильному порядку. Функція, що викликає, вважає, що регістр, який зберігається викликаною функцією, залишається незмінним після виклику функції. Регістри аргументів можуть бути змінені під час виконання функції, тому відповідальність за їх збереження і відновлення зі стека залишається на функції, що викликає, якщо в цьому є необхідність. Тимчасові регістри також

можуть змінюватися в функції, що викликається, тому при необхідності функція, що викликає, повинна їх зберігати і відновлювати. Цілком очевидно, що регістр rax, що містить значення, яке повертається, повинен зберігатися і відновлюватися функцією, яка викликає.

Таблиця 5.3 – Угоди про виклик функцій

| Регістр | Використання | Хто зберігає |
|---|---|---|
| rax | Значення, що повертається | Функція, яка викликає |
| rbx | Зберігається функцією, яка викликає | Функція, яка викликає |
| rcx | 4-й аргумент | Функція, яка викликає |
| rdx | 3-й аргумент | Функція, яка викликає |
| rsi | 2-й аргумент | Функція, яка викликає |
| rdi | 1-й аргумент | Функція, яка викликає |
| rbp | Зберігається функцією, яка викликає | Функція, яка викликає |
| rsp | Вказівник стеку | Функція, яка викликає |
| r8 | 5-й аргумент | Функція, яка викликає |
| r9 | 6-й аргумент | Функція, яка викликає |
| r10 | Тимчасовий | Функція, яка викликає |
| r11 | Тимчасовий | Функція, яка викликає |
| r12 | Зберігається функцією, яка викликає | Функція, яка викликає |
| r13 | Зберігається функцією, яка викликає | Функція, яка викликає |
| r14 | Зберігається функцією, яка викликає | Функція, яка викликає |
| r15 | Зберігається функцією, яка викликає | Функція, яка викликає |
| xmm0 | 1-й аргумент та значення, що повертається | Функція, яка викликає |
| xmm1 | 2-й аргумент та значення, що повертається | Функція, яка викликає |
| xmm2-xmm7 | Аргументи | Функція, яка викликає |
| xmm8-xmm15 | Тимчасові | Функція, яка викликає |

Проблеми можуть виникати при зміні існуючої функції, коли в ній починається використання регістра, що зберігається функцією, яка викликає. Якщо не додати інструкції збереження та відновлення цього регістру в функції яка викликає, то можна отримати непередбачувані результати.

Регістри, що зберігаються функцією, також називаються неруйнівними, або довготривалими (nonvolatile). Регістри, які повинна зберігати функція, що викликає, ще називаються руйнованими, або короткочасними (volatile).

Всі xmm регістри можуть бути змінені функцією, що викликається, тому при необхідності функція, що викликає, відповідає за їх збереження і відновлення.

Зрозуміло, якщо є впевненість у тому, що не будуть використовуватися регістри, що змінюються, то можна пропустити інструкції їх збереження.

Але якщо в майбутньому код зміниться, можливо, виникнуть проблеми, якщо почати використовувати ці регістри без збереження їх вмісту. Зауваження:

syscall – це також функція, яка змінює регістри, тому треба уважно стежити за тим, що вона робить [29].

**5.3.7 Введення та виведення з консолі.** Вже відомо, як виконується виведення у консолі з використанням системних викликів або зовнішньої функції printf. У цьому розділі знову будуть застосовуватися системні виклики, але не тільки для виведення на екран, а ще й для введення з клавіатури.

У лістингу 5.16 показаний приклад вихідного коду без використання зовнішніх функцій та системних викликів.

Лістинг 5.16 – Програма console1.asm

```
; console1.asm
section .data
msg1              db "Hello, World!",10,0
msg1len           equ $-msg1
msg2              db "Your turn: ",0
msg2len           equ $-msg2
msg3              db "You answered: ",0
msg3len           equ $-msg3
inputlen          equ 10      ; Довжина буфера введення.
section .bss
input resb inputlen+1         ; Забезпечення місця для завершального 0.
section .text
global main
main:
push rbp
mov rbp,rsp
mov rsi, msg1                 ; Виведення першого рядка.
mov rdx, msg1len
call prints
mov rsi, msg2                 ; Виведення другого рядка, без NL.
mov rdx, msg2len
call prints
mov rsi, input                ; Адреса буфера введення inputbuffer.
mov rdx, inputlen             ; Довжина буфера введення inputbuffer.
call reads                    ; Очікування введення.
mov rsi, msg3                 ; Виведення третього рядка.
mov rdx, msg3len
call prints
mov rsi, input          ; Виведення вмісту введення inputbuffer.
mov rdx, inputlen       ; Довжина буфера введення inputbuffer.
call prints
leave
ret
;------------------------------------------------ ---
prints:
push rbp
mov rbp, rsp
; rsi містить адресу рядка.
; rdx містить довжину рядка.
mov rax, 1              ;1=запис
mov rdi, 1              ; 1 = stdout – стандартне виведення.
syscall
leave
ret
;------------------------------------------------ ---
reads:
push rbp
mov rbp, rsp
```

```
; rsi містить адресу буфера введення inputbuffer.
; rdi містить довжину вводу буфера введення inputbuffer.
mov rax, 0              ;0=читання
mov rdi, 1              ;1 = stdin - стандартне введення.
syscall
leave
ret
```

Це не дуже складна програма, в ній створюється буфер введення з ім'ям input для зберігання символів, що вводяться. Крім того, визначається довжина цього буфера в змінній inputlen. Після виведення кількох вітальних повідомлень викликається функція reads, яка приймає символи, що вводяться з клавіатури, і повертає їх у програму, що викликає, після натискання клавіші Enter. Потім програма, що викликає, використовує функцію prints для виведення раніше введених символів. Але тут виникають деякі проблеми. Для буфера введення зарезервовано 10 байт.

Ця програма приймає лише 10 символів і не знає, що робити з іншими «зайвими» символами, тому повертає їх до операційної системи. Операційна система намагається визначити їх зміст та інтерпретує ці символи як команди, але не знаходить відповідні файли чи імена вбудованих команд, тому виводить повідомлення про помилки.

Результат невтішний, але він навіть гірший, ніж здається на перший погляд. Такий спосіб обробки виводу може стати причиною серйозної вразливості у системі забезпечення безпеки, через яку хакер може зламати програму та отримати доступ до операційної системи.

**Обробка переповнень.** У лістингу 5.17 показано іншу версію програми, в якій ведеться підрахунок символів, а зайві символи просто відкидаються. Додаткова умова: дозволено вводити лише алфавітні символи в нижньому регістрі, від a до z.

Лістинг 5.17 - Програма console2.asm

```
; console2.asm
section .data
msg1        db "Hello, World!",10,0
msg2        db "Your turn (тільки a-z): ",0
msg3        db "You answered: ",0
inputlen    equ 10; Довжина буфера введення.
NL          db 0xa
section .bss
input       resb inputlen+1   ; Забезпечення місця для завершального 0.
section .text
global main
main:
push rbp
mov rbp,rsp
mov rdi, msg1     ; Виведення першого рядка.
call prints
mov rdi, msg2     ; Виведення другого рядка, без NL.
call prints
mov rdi, input    ; Адреса буфера введення inputbuffer.
mov rsi, inputlen ; Довжина буфера введення inputbuffer.
call reads
; Очікування введення.
mov rdi, msg3     ; Виведення третього рядка та додавання введеного рядка.
```

```
call prints
mov rdi, input    ; Виведення вмісту буфера введення inputbuffer.
call prints
mov rdi,NL        ; Виводить символ переходу на новий рядок NL.
call prints
leave
ret
;------------------------------------------------ ---------
prints:
push rbp
mov rbp, rsp
push r12          ; Регістр, що зберігається функцією, яка викликається.
; Підрахунок символів.
xor rdx, rdx      ; Довжина rdx.
mov r12, rdi
.lengthloop:
cmp byte [r12], 0
je
.lengthfound
inc rdx
inc r12
jmp .lengthloop
.lengthfound:
; Виведення рядка, його довжина в rdx.
cmp rdx, 0
; Рядок відсутній (довжина 0).
je
.done
mov rsi,rdi       ; rdi містить адресу рядка.
mov rax, 1        ; 1 = запис.
mov rdi, 1        ; 1 = stdout – стандартний висновок.
syscall
.done:
pop r12
leave
ret
;------------------------------------------------ ---------
reads:
section .data
section .bss
.inputchar
resb
1
section .text
push rbp
mov rbp, rsp
push r12          ; Регістр, що зберігається функцією, яка викликається.
push r13          ; Регістр, що зберігається функцією, яка викликається.
push r14          ; Регістр, що зберігається функцією, яка викликається.
mov r12, rdi      ; Адреса буфера введення inputbuffer.
mov r13, rsi      ; Максимальна довжина r13.
xor r14, r14
; Лічильник символів.
.readc:
mov rax, 0              ; Читання.
mov rdi, 1              ; stdin – стандартне введення.
lea rsi, [.inputchar]   ; Адресу джерела введення.
mov rdx, 1              ; Число символів, що зчитуються.
syscall
mov al, [.inputchar]    ; Введено символ NL?
cmp al, byte[NL]
je
.done
; NL - кінець введення.
cmp al, 97              ; Код символу менший за a?
```

```
jl .readc
; Відкинути символ.
cmp al, 122             ; Код символу більший за z?
jg .readc
; Відкинути символ.
inc r14                 ; Збільшити лічильник символів на 1.
cmp r14, r13
ja
.readc
; Максимальне наповнення буфера, відкинути зайве.
mov byte [r12], al      ; Зберегти символ у буфері.
inc r12
; Перемістити вказівник на наступний символ у буфері.
jmp .readc
. done:
inc r12
mov byte [r12],0  ; Додати завершальний 0 до буфера введення inputbuffer.
pop r14           ; Регістр, що зберігається функцією, яка викликається.
pop r13           ; Регістр, що зберігається функцією, яка викликається.
pop r12           ; Регістр, що зберігається функцією, яка викликається.
leave
ret
```

Функція prints змінена: спочатку вона підраховує кількість символів, що виводяться, тобто рахунок ведеться до тих пір, поки не зустрінеться байт 0. Після визначення довжини prints виводить рядок за допомогою syscall.

Функція reads очікує введення одного символу та перевіряє, чи є він символом переходу на новий рядок (NL). Якщо отримано символ переходу на новий рядок, читання символів з клавіатури припиняється. Регістр r14 з тримає лічильник символів, що вводяться. Функція стежить за тим, щоб кількість введених символів не перевищувала довжину буфера введення inputlen. Якщо ця кількість менша за довжину буфера, то символ додається в буфер input. Якщо довжина inputlen перевищена, то символ відкидається, але читання з клавіатури продовжується. Введено додаткову вимогу: ASCII код символу повинен бути не менше 97 (a) і не більше 122 (z). Ця вимога забезпечує прийом тільки алфавітних символів у нижньому регістрі. Зверніть увагу на збереження та відновлення регістрів, що зберігаються викликаною функцією (callee saved), – регістр r12 використовується в обох функціях prints та reads. У цьому прикладі відсутність інструкцій збереження таких (callee saved) регістрів не повинно призводити до проблем, але легко уявити собі ситуацію, коли одна з функцій викликає іншу, а функція, що викликається звертається до наступної, тоді можуть виникати проблеми [29].

**5.3.8 Командний рядок.** Іноді необхідно при запуску програми з командного рядка вказати аргументи, які використовуватиме ця програма. Це може виявитися корисним при створенні власних інструментальних засобів командного рядка. Системні адміністратори постійно застосовують інструменти командного рядка, тому що, як правило, такі інструменти працюють швидше в руках досвідченого користувача.

Доступ до аргументів командного рядка. У прикладі програми у лістингу 5.18 показано, як можна отримати доступ до аргументів командного рядка в

асемблерній програмі. Тут розглядаються найпростіші дії: пошук аргументів та його виведення.

Лістинг 5.18 – Програма cmdline.asm

```
;cmdline.asm
extern printf
section .data
msg         db "The command and arguments: ",10,0
fmt         db "%s",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp, rsp
mov r12, rdi      ; Кількість аргументів.
mov r13, rsi      ; Адреса масиву аргументів.
; Виведення заголовка.
mov rdi, msg
call printf
mov r14, 0
; Виведення імені команди та аргументів.
.ploop:           ; Цикл проходу за масивом аргументів та їх виведення.
mov rdi, fmt
mov rsi, qword [r13+r14*8]
call printf
inc r14
cmp r14, r12      ; Чи досягнуто максимальної кількості аргументів?
jl .ploop
leave
ret
```

За виконання цієї програми кількість аргументів, включаючи ім’я самої програми, зберігається у регістрі rdi. Регістр rsi містить адресу пам’яті масиву, що містить адреси аргументів командного рядка, причому першим аргументом є ім’я самої програми. Використання регістрів rdi та rsi відповідає угодам про виклики функцій. Нагадаю, що ми працюємо в системі Linux та використовуємо угоди про виклики функцій System V AMD64 ABI. На інших платформах, наприклад у Windows, застосовуються інші угоди про виклики функцій. Інформація з цих регістрів копіюється, тому що rdi та rsi пізніше будуть використовуватися для функції printf.

У цьому коді існує цикл для проходу масивом аргументів до тих пір доки не буде досягнуто максимальної кількості аргументів. У циклі .ploop регістр r13 свідчить про масив аргументів. Регістр r14 використовується як лічильник аргументів. На кожній ітерації циклу обчислюється наступна адреса ного аргументу і записується в rsi. Число 8 у виразі qword [r13+r14*8] відповідає довжині зазначених адрес: 8 байт × 8 біт = 64-х бітовій адресі.

Регістр r14 кожної ітерації циклу порівнюється з регістром r12, що містить загальну кількість аргументів [29].

**5.3.9 Рядки в асемблері.** З погляду людини рядки зазвичай видаються у вигляді послідовності символів, що утворюють слова та фрази, які можна зрозуміти. Але у мові асемблера будь-який список чи масив безперервних

локацій пам’яті вважається рядком незалежно від того, чи може зрозуміти її людина.

Асемблер надає кілька потужних інструкцій для обробки таких блоків даних ефективним способом. У наведених нижче прикладах використовуються символи, що зчитуються, але слід пам’ятати, що насправді асемблер не дбає про те, щоб символи були зчитаними. Тут буде показано, як можна переміщати рядки до інших локацій, як сканувати та порівнювати рядки.

У лістингу 5.19 показаний приклад вихідного коду.

Лістинг 5.19 – Програма move_strings.asm

```
; move_strings.asm
%macro prnt 2
mov rax, 1        ; 1 = запис.
mov rdi, 1        ; 1 = стандартний потік виведення stdout.
mov rsi, %1
mov rdx, %2
syscall
mov rax, 1
mov rdi, 1
mov rsi, NL
mov rdx, 1
syscall
%endmacro
section .data
length      equ 95
NL          db 0xa
string1     db "my_string of ASCII:"
string2     db 10,"my_string of zeros:"
string3     db 10,"my_string of ones:"
string4     db 10,"again my_string of ASCII:"
string5     db 10,"copy my_string to other_string:"
string6     db 10,"reverse copy my_string to other_string:"
section .bss
my_string resb length
other_string resb length
section .text
global main
main:
push rbp
mov rbp, rsp
;------------------------------------------------ -
; Заповнення рядка видимими ascii-символами.
prnt string1, 18
mov rax, 32
mov rdi, my_string
mov rcx, length
str_loop1: mov byte[rdi], al ; Найпростіший метод.
inc rdi
inc al
loop str_loop1
prnt my_string, length
;------------------------------------------------ -
; Заповнення рядка ascii символами 0.
prnt string2, 20
mov rax,48
mov rdi, my_string
mov rcx, length
str_loop2: stosb ; Тут не потрібна інструкція inc rdi.
loop str_loop2
prnt my_string, length
```

```
;------------------------------------------------ -
; Заповнення рядка ascii символами 1.
prnt string3,19
mov rax, 49
mov rdi,my_string
mov rcx, length
rep stosb; Тут уже не потрібна інструкція inc rdi та не потрібен цикл.
prnt my_string,length
;------------------------------------------------ -
; Повторне заповнення рядка видимими ascii-символами.
prnt string4, 26
mov rax, 32
mov rdi, my_string
mov rcx, length
str_loop3: mov byte[rdi], al ; Найпростіший метод.
inc rdi
inc al
loop str_loop3
prnt my_string, length
;------------------------------------------------ -
; Копіювання рядка my_string в інший рядок other_string.
prnt string5, 32
mov rsi, my_string ; Джерело в RSI.
mov rdi, other_string ; Ціль в rdi.
mov rcx, length
rep movsb
prnt other_string,length
;------------------------------------------------ -
; Копіювання у зворотному порядку рядка my_string в інший рядок
other_string.
prnt string6,40
mov rax, 48
; Очищення рядка other_string.
mov rdi,other_string
mov rcx, length
rep stosb
lea rsi,[my_string+length-4]
lea rdi,[other_string+length]
mov rcx, 27 ; Копіювання лише 27-1 символів.
itd ; std встановлює прапор DF, cld скидає прапор DF.
rep movsb
prnt other_string, length
leave
ret
```

У цій програмі застосовується макрос для виведення рядків, але можна було б застосувати функцію мови C printf.

Ми починаємо зі створення рядка, що містить 95 видимих символів з таблиці ASCII, – код першого символу рядка 32 (пробіл), код останнього символу 126 (~). У цих знаках немає нічого незвичайного. Спочатку виводиться заголовок, потім ASCII код першого символу записується в регістр rax, а регістр rdi вказує адресу в пам’яті рядка my_string. Потім довжина рядка, що формується, передається в регістр rcx для використання в циклі. У цьому циклі один ASCII код копіюється з регістру al у рядок my_string, потім виконується перехід до наступного ASCII коду, який записується за черговою адресою пам’яті my_string і т. д. Після завершення циклу виводиться отриманий рядок.

У наступній частині програми весь вміст рядка my_string повністю замінюється на символи 0 (ASCII код 48). Довжина рядка знову записується у регістр rcx для організації циклу. Далі використовується інструкція stosb для

запису символів 1 (ASCII код 49) у рядок my_string. Для інструкції stosb необхідна тільки початкова адреса рядка в регістрі rdi і символ у регістрі rax, і stosb автоматично переходить до наступної адреси пам'яті на кожному кроці циклу. У цьому випадку можна не дбати про збільшення вмісту регістра rdi.

Далі в програмі вживаються додаткові заходи, за винятком циклу з допомогою регістра rcx. Застосовується інструкція rep stosb для повторення stosb задану кількість разів. Кількість повторень міститься у регістрі rcx. Це дуже ефективний спосіб ініціалізації пам'яті.

Строго кажучи, тут будуть копіювати блоки пам'яті, а не переміщувати їх вміст. Спочатку рядок знову ініціалізується кодами ASCII видимих символів. Можна було б оптимізувати цю частину коду, застосувавши макрос або функцію замість простого повторення фрагмента вихідного коду. Потім починається копіювання рядка/блоку пам'яті: з my_string до other_string. Адреса рядки джерела міститься в регістр rsi, адреса цільового рядка записується в регістр rdi. Це легко запам'ятати, тому що буква s в імені регістра rsi позначає source (джерело), а літера d у імені регістра rdi – destination (мета). Потім виконується інструкція rep movsb і все зроблено. Інструкція rep продовжує копіювання доти, доки вміст rcx не стане рівним 0.

В останній частині програми виконується переміщення (копіювання) вмісту пам'яті у зворотному порядку. Ця концепція може здатися дещо складнішою, тому нижче пояснюються деякі складності. При використанні інструкції movsb враховується значення DF. Якщо DF=0, то значення в регістрах rsi та rdi збільшуються на 1, вказуючи на наступну старшу адресу пам'яті. Якщо DF=1, то значення rsi і rdi зменшуються на 1, вказуючи на наступну молодшу адресу пам'яті. Це означає, що в цьому прикладі при DF=1 необхідно, щоб регістр rsi вказував на найстаршу адресу пам'яті в блоці, що скопійована, та зменшувався, починаючи з цієї адреси. При копіюванні передбачається «прохід у зворотному порядку», тобто зменшення rsi та rdi на кожному кроці циклу. Будьте уважні: зменшуються обидва регістри rsi та rdi, оскільки неможливо використовувати прапор DF для збільшення одного регістру та зменшення іншого (тобто для реверсування рядка). У цьому прикладі копіюється не весь рядок повністю, а лише алфавітні символи в нижньому регістрі, які розміщуються у старших адресах пам'яті у цільовому рядку.

Інструкція lea rsi, [my_string+length-4] завантажує ефективну адресу рядка my_string у регістр rsi та пропускає чотири символи, які не є алфавітними. Прапор DF можна встановити в 1 інструкції std і скинути в 0 інструкцією cld. Потім викликається потужна інструкція rep movsb і завдання виконано.

Чому в регістр rcx записується значення 27, хоча обробляються 26 символів? Виявляється, інструкція rep зменшує rcx на 1 перед виконанням будь-яких операцій усередині циклу. Це можна перевірити за допомогою debuger-a [29].

## Лабораторна робота № 16-17
## Програмування асемблері в 64-х бітному режимі

Порядок виконання роботи.

1. Завантажити та встановити Oracle VirtualBox за посиланням: https://download.virtualbox.org/
2. Завантажити Kali Linux (або інший дистрибутив з гілки Debian за вибором студента) за посиланням https://cdimage.kali.org/kali-2022.4/kali-linux-2022.4-virtualbox-amd64.7z
3. Запустити Kali Linux (логін: kali, пароль: kali).
4. Встановити наступні програмні продукти:
   a. оновити репозиторії командою sudo apt-get update;
   b. gcc (sudo apt-get install gcc);
   c. gdb (sudo apt-get install gdb);
   d. make (sudo apt-get install make);
   e. nasm (sudo apt-get install build-essential nasm);
   f. SASM (sudo apt-get install sasm).
5. Виконати налаштування SASM (рис. 5.26)

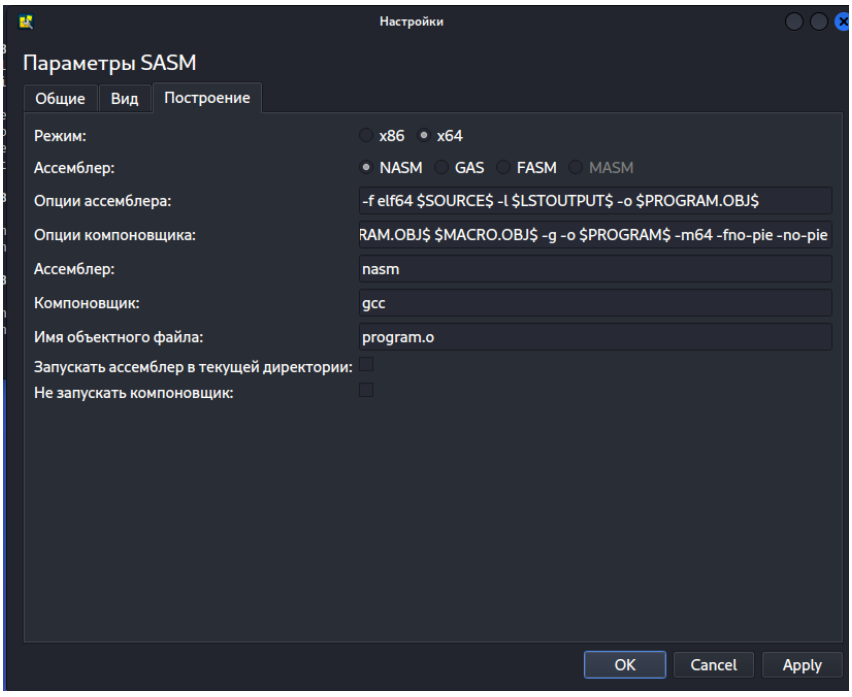



Рисунок 5.26 – Налаштування SASM

6. Виконати в SASM компіляцію наведеного прикладу:

```
; fcalc.asm
extern printf
section .data
number1             dq 9.0
number2             dq 73.0
fmt                 db "Numbers are %f and %f",10,0
fmt float           db "%s %f",10,0
f_sum               db "Це float sum of %f and %f is %f",10,0
f_dif               db "The float difference of %f and %f is %f",10,0
f_mul               db "The float product %f i %f e %f",10,0
```

```
f_div               db "The float division of %f by %f is %f",10,0
f_sqrt              db "The float squareroot of %f is %f",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
; Виведення чисел.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, fmt
mov rax, 2                ; Два числа з плаваючою точкою.
call printf
; Обчислення суми.
movsd xmm2, [number1]     ; Число з плаваючою точкою подвійної точності xmm.
addsd xmm2, [number2]     ; Додавання з іншим числом подвійної точності,
                          ;результат xmm.
; Виведення результату.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, f_sum
mov rax, 3                ; Три числа з плаваючою точкою.
call printf
; Обчислення різниці.
movsd xmm2, [number1]     ; Число з плаваючою точкою подвійної точності xmm.
subsd xmm2, [number2]     ; Віднімання іншого числа подвійної точності з xmm.
; Виведення результату.
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi, f_dif
mov rax, 3                ; Три числа з плаваючою точкою.
call printf
; множення.
movsd xmm2, [number1]     ; Число з плаваючою точкою подвійної точності xmm.
mulsd xmm2, [number2]     ; Збільшення заданого числа на xmm.
; Виведення результату.
mov rdi,f_mul
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,3                 ; Три числа з плаваючою точкою.
call printf
; Ділення.
movsd xmm2, [number1]     ; Число з плаваючою точкою подвійної точності xmm.
divsd xmm2, [number2]     ; Ділення xmm0.
; Виведення результату.
mov rdi,f_div
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,1                 ; Одне число з плаваючою точкою.
call printf
; Обчислення квадратного кореня.
sqrtsd xmm1, [number1]    ; Квадратний корінь подвійної точності xmm.
; Виведення результату.
mov rdi, f_sqrt
movsd xmm0, [number1]
mov rax, 2                ; Два числа з плаваючою точкою.
call printf
; Вихід з програми.
mov rsp, rbp
pop rbp                   ; Операція, обернена операції push на початку програми.
ret
```

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 | Функція | Варіант | Функція |
|---|---|---|---|
| 1 | $y = 3,5x^2 + 7,2x + 24$ | 14 | $y = \sqrt{14x^2 + 5,6x - 20}$ |
| 2 | $a_n = 2,5n^2 + 5,3n - 21$ | 15 | $y = \frac{20x - 15}{5x^2 - 8,5x + 15,2}$ |
| 3 | $y = \frac{2500}{2x^2 + 3,7}$ | 16 | $a_n = \frac{4n^3 - 1}{n + 5,4}$ |
| 4 | $y = \sqrt{14x^2 + 5,6x}$ | 17 | $y = \frac{x - 7}{\sqrt{x^2 + 20x - 24}}$ |
| 5 | $y = \frac{20x}{5x^2 - 8,5}$ | 18 | $y = \frac{1024 - 2x^2}{3,2x^2 - 25}$ |
| 6 | $a_n = \frac{4,5^n}{n + 5,1}$ | 19 | $a_n = \frac{418 + n}{2n^2 + 2,3}$ |
| 7 | $y = \frac{x - 7}{\sqrt{x^2 + 20}}$ | 20 | $y = \frac{3000 - 2x^2 + 2,5x}{x^2 + 3,6x - 7,5}$ |
| 8 | $y = \frac{1024}{3,2x^2 - 25}$ | 21 | $y = \frac{3,5x^2 + 7,2x + 24}{x^2 - 23,4}$ |
| 9 | $a_n = \frac{418}{2n^2 + 7,3}$ | 22 | $a_n = \frac{2,1n^2 - 2,3n - 21}{n + 1,4}$ |
| 10 | $y = \frac{3000 + x}{x^2 + 2,6x - 7,5}$ | 23 | $y = \frac{4000}{2x^2 + 3,7x - 4,3}$ |
| 11 | $y = 3,5x^2 + 7,2x - 2,2$ | 24 | $y = \sqrt{14x^2 + 5,6x - 2,4}$ |
| 12 | $a_n = 2,5n^2 + 5,3n + 5$ | 25 | $y = \frac{1124,4 - x}{3,2x^2 - 25}$ |
| 13 | $y = \frac{2500,12}{2x^2 + 3,7x - 8,1}$ | | |

**ДОДАТОК 1**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ПОЛТАВСЬКА ПОЛІТЕХНІКА
ІМЕНІ ЮРІЯ КОНДРАТЮКА»

Навчально-науковий інститут інформаційних технологій та робототехніки
Кафедра комп'ютерних та інформаційних технологій і систем

# Лабораторна робота № 9

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

Варіант – 6

**Виконала:**
*студентка 101-ТН*
*Івінська Катерина Дмитрівна*
**Перевірив:**
*Демиденко Максим Ігорович*

Полтава 2023

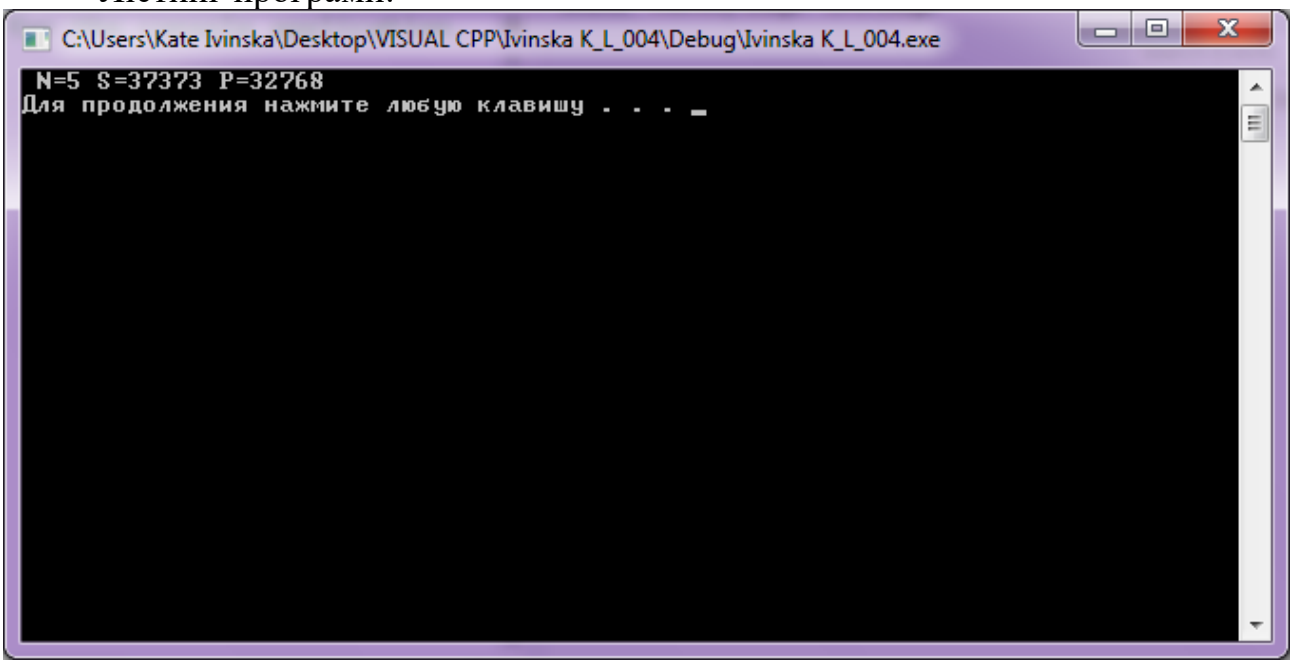
## Основна частина

Визначити номер ($n$) елемента прогресії $a_n = 8n - 5$, 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:



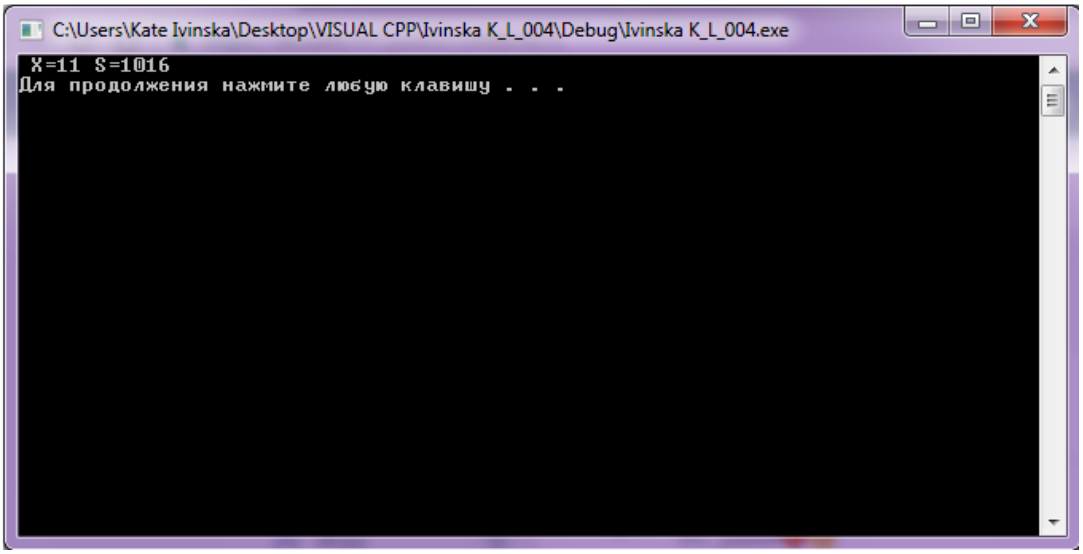

## Індивідуальне завдання

Знайти ціле значення аргументу, при якому функція $y = 9x^2 - 8x + 15$ стане більшою за 1000.

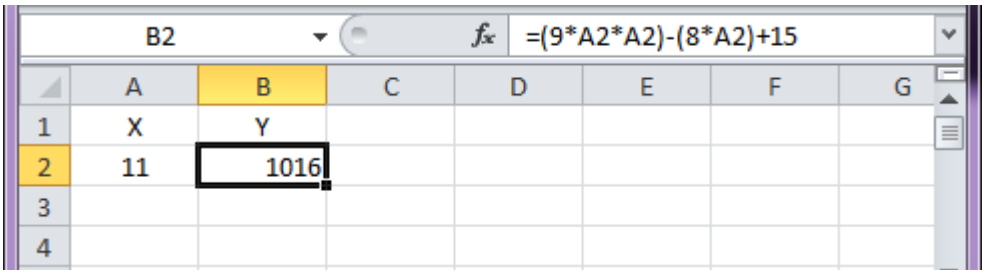
```
#include"stdafx.h"
#include<iostream>
usingnamespace std;
void main() // початок програми мовою с++
```

```
{
        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++, 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. Structured Computer Organization 6th Edition/ Andrew Tanenbaum, Todd Austin, Pearson 2012 – 808 p.
11. Історія розвитку обчислювальної техніки [Електронний ресурс] – http://informatics.dp.ua/istoriya-rozvytku-obchyslyuvalnoyi-tekhniky/
12. Комп’ютерна схемотехніка та архітектура комп’ютерів. Конспект лекцій для студентів спеціальності 5.05010101 «Обслуговування програмних систем і комплексів» денної форми навчання / Пастушок І.М. – Ковель: КПЕК Луцького НТУ, 2014. – 186 с.
13. Грід [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D2%90%D1%80%D1%96%D0%B4
14. Хмарні обчислення [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A5%D0%BC%D0%B0%D1%80%D0%BD%D

1%96_%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F
15. Форм-фактор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC-%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80
16. Процессор Intel® Core™ i5-12400F [Електронний ресурс] – Режим доступу до ресурсу: https://ark.intel.com/content/www/ru/ru/ark/products/134587/intel-core-i512400f-processor-18m-cache-up-to-4-40-ghz.html
17. Тип пам'яті SSD накопичувачів [Електронний ресурс] – Режим доступу до ресурсу: https://2400.com.ua/ua/a434821-tip-pamyati-ssd.html
18. Serial ATA [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Serial_ATA
19. M.2 [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/M.2
20. Serial Attached SCSI (SAS) [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Serial_Attached_SCSI
21. Інтерфейси моніторів (VGA, DVI, HDMI, DisplayPort) [Електронний ресурс] – Режим доступу до ресурсу: https://boss-pc.com.ua/interfeysi-monitoriv-vga-dvi-hdmi-displayport/
22. Оптичний привод [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D1%87%D0%BD%D0%B8%D0%B9_%D0%BF%D1%80%D0%B8%D0%B2%D0%BE%D0%B4
23. USB-флеш-накопичувач [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/USB-%D1%84%D0%BB%D0%B5%D1%88-%D0%BD%D0%B0%D0%BA%D0%BE%D0%BF%D0%B8%D1%87%D1%83%D0%B2%D0%B0%D1%87
24. Рідкокристалічний дисплей [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A0%D1%96%D0%B4%D0%BA%D0%BE%D0%BA%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BB%D1%96%D1%87%D0%BD%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BF%D0%BB%D0%B5%D0%B9
25. Принтер [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%82%D0%B5%D1%80
26. IA-32 Intel® Architecture Software Developer's Manual. Vol. 1. Basic architecture. Intel Corporation, 2002.
27. IA-32 Intel® Architecture Software Developer's Manual. Vol. 3. System programming guide. Intel Corporation, 2002.
28. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume1: Basic Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://cdrdv2.intel.com/v1/dl/getContent/671436
29. Beginning x64 Assembly Programming From Novice to AVX Professional / Jo Van Hoey - Apress Berkeley, CA, 2019 – 413p