Олексій Шелуха

Основи архітектури комп'ютера

Навчальний посібник

Міністерство освіти і науки України
Державний університет «Житомирська політехніка»

Олексій Шелуха

ОСНОВИ АРХІТЕКТУРИ КОМП’ЮТЕРА

Навчальний посібник

Житомир
2026

УДК 004.2
Ш42

Рекомендовано Вченою радою
Державного університету «Житомирська політехніка»
(протокол № 7 від 27 квітня 2026 року)

Рецензенти:

Ю. О. Гунченко – доктор технічних наук, професор, завідувач кафедри комп'ютерних систем та технологій Одеського національного університету ім. І.І. Мечнікова;

С. В. Лазаренко – доктор технічних наук, професор, професор кафедри технічного захисту інформації Державного університету «Київський авіаційний інститут»;

В. В. Воротніков – доктор технічних наук, доцент, професор кафедри комп'ютерної інженерії та кібербезпеки Державного університету «Житомирська політехніка».

Шелуха О. О.

Ш42 Основи архітектури комп'ютера : навч. посібник / О. О. Шелуха. – Електронні дані. – Житомир : Державний університет «Житомирська політехніка», 2026. – 208 с.

ISBN 978-966-683-731-1

Навчальний посібник присвячено історичному розвитку комп'ютерної техніки та процесів, що покладені в основи її функціонування, зокрема системам числення, комп'ютерній логіці, фізичній реалізації обчислюючих елементів та роботі з асемблером.

Матеріал навчального посібника структуровано відповідно до логіки вивчення дисципліни та спрямовано на формування цілісного уявлення про принципи побудови і функціонування архітектури комп'ютера.

Посібник призначено для здобувачів вищої освіти в галузі інформаційних технологій, зокрема за спеціальностями комп'ютерна інженерія, кібербезпека та захист інформації, комп'ютерні науки, інформаційні системи та технології, а також для викладачів і всіх, хто цікавиться питаннями функціонування та побудови комп'ютера.

УДК 004.2

ISBN 978-966-683-731-1

ЗМІСТ

ПЕРЕДМОВА

У сучасному світі інформація є одним із важливіших ресурсів, і розуміння принципів її обробки, передавання та захисту є запорукою успішного функціонування держави, науки та бізнесу. Важко уявити будь-яку сферу людської діяльності без застосування комп’ютерної техніки: від смартфонів у кишені до гігантських мейнфреймів і суперкомп’ютерів, що моделюють кліматичні зміни. Проте, щоб ці пристрої могли виконувати складні алгоритми штучного інтелекту, обробляти графіку чи забезпечувати кібербезпеку, вони мають бути побудовані за певними правилами, які ми називаємо архітектурою комп'ютера.

Архітектура комп’ютера є однією з основних концепцій у сфері інформатики та обчислювальної техніки, що визначає принципи побудови систем та правила взаємодії апаратного і програмного забезпечення. У сучасному цифровому світі розуміння внутрішньої логіки комп’ютерної техніки стає необхідною умовою для формування кваліфікованого ІТ-фахівця.

Цей навчальний посібник підготовлено з метою систематизованого викладу фундаментальних засад функціонування обчислювальних систем. Матеріал охоплює шлях від елементарних одиниць інформації до складних архітектурних рішень, дозволяючи студентам сформувати цілісне уявлення про те, як абстрактні одинички та нулики перетворюються на реальні обрахунки за допомогою електронних компонентів, та інструкції мікропроцесорів.

Структура посібника відображає логіку поетапного занурення у предмет:

Розділ 1 знайомить із концепцією архітектури, розмежовує її з поняттям організації комп'ютера та розкриває роль міжнародних стандартів, що забезпечують сумісність сучасного обладнання.

Розділи 2 та 3 закладають математичний і логічний фундамент, описуючи системи числення, представлення різних типів даних у пам'яті та принципи булевої алгебри, на яких будуються логічні вентилі та арифметико-логічні пристрої (ALU).

Розділ 4 присвячено еволюції обчислювальної техніки та детальному аналізу компонентів сучасних систем: від класичної моделі фон Неймана до інтегрованих систем-на-кристалі (SoC), багатоядерних процесорів, ієрархії пам'яті та швидкісних шин взаємодії.

Розділ 5 розкриває особливості низькорівневого програмування мовою асемблера, роботу з регістрами, стеком та математичним співпроцесором, що є критично важливим для оптимізації коду, розробки драйверів та завдань кібербезпеки.

Зміст посібника орієнтований на забезпечення навчальних потреб здобувачів вищої освіти у процесі вивчення дисциплін, пов’язаних із комп'ютерною інженерією, програмуванням та захистом інформації. Знання архітектурних особливостей дозволить майбутнім фахівцям не лише розуміти принципи роботи техніки, а й ефективно використовувати ресурси процесора, проводити реверс-інжиніринг та створювати високопродуктивні інформаційні системи.

Автор має надію, що матеріал викладений в даному посібнику допоможе здобувачеві поєднати фундаментальні знання з практикою та стане надійною основою для подальшого вивчення сучасних та перспективних обчислювальних технологій.

РОЗДІЛ 1. ВВЕДЕННЯ В АРХІТЕКТУРУ КОМП'ЮТЕРА

1.1. Концепція архітектури комп'ютера.

Сучасній людині важко уявити своє життя без застосування тієї чи іншої комп’ютерної техніки. Кожен з нас має у використанні принаймні один комп’ютерний пристрій, такий як смартфон, ноутбук або стаціонарний комп’ютер (рис.1.1). Кожен з цих пристроїв має свої функції та задачі, повний перелік яких обмежується лише фантазією та фінансовими можливостями користувача. Проте для виконання цих задач, взаємодію комп’ютерних систем та компонентів між собою вони будуються за певними правилами, яку можна назвати архітектурою.

Рисунок 1.1 – Комп’ютерні та периферійні пристрої

Архітектура комп’ютера є однією з основних концепцій у сфері інформатики та комп’ютерної техніки. Вона визначає, як побудовані комп’ютери, які принципи лежать в основі їхньої роботи, як взаємодіють апаратне і програмне забезпечення. Це поняття охоплює опис функцій, можливостей та організації основних компонентів системи: процесора, пам’яті, пристроїв введення/виведення та засобів їхньої комунікації. Розуміння архітектури комп’ютера є критично важливим для всіх, хто працює у сфері програмування, системного адміністрування, кібербезпеки, розробки та створення високопродуктивних інформаційних систем тощо.

Коли ми говоримо про архітектуру комп’ютера, важливо чітко розмежовувати два основні поняття: "архітектура" і "організація". Архітектура визначає, що може виконувати комп’ютер, тобто набір його функцій, інструкцій, які виконує процесор, і загальну структуру системи. Натомість організація стосується того, як ці функції реалізовані фізично, наприклад, яким чином спроєктовані мікросхеми або якими засобами побудована системна плата.

Сучасна архітектура комп’ютера базується на моделі, яку запропонував Джон фон Нейман у 1945 році [1]. Вона включає ключові принципи: програми і дані зберігаються у спільній пам’яті, а інструкції виконуються послідовно. Основними компонентами цієї архітектури є:

1. **Процесор** – виконує обчислення та керує всією системою.

2. **Пам’ять** – зберігає програми й дані.

3. **Пристрої введення/виведення** – забезпечують обмін інформацією між комп’ютером і зовнішнім середовищем.

4. **Системна шина** – передає дані між цими компонентами.

Не зважаючи на простоту та зручність Архітектура фон Неймана має одне суттєве обмеження: «вузьке місце» шини пам’яті (1.2, а). Усі команди та дані повинні проходити через єдиний канал зв’язку (шину), що уповільнює обчислення. Це явище отримало назву "bottleneck" (укр. дослівно «пляшкове горлечко»), і саме воно стало стимулом для розробки інших підходів, зокрема гарвардської архітектури (рис. 1.2, б).

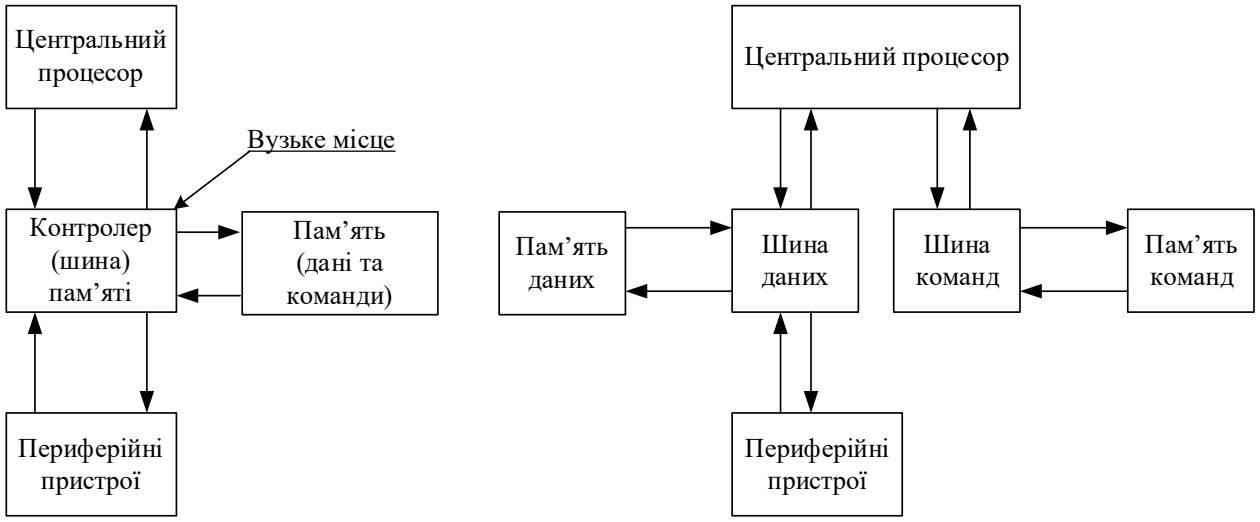

Рисунок 1.2 – Порівняння фон-нейманівської (а) та гарвардської архітектур (б)

Сучасна архітектура виходить далеко за рамки традиційних моделей. Наприклад, багатоядерні процесори дозволяють обробляти кілька завдань одночасно, а гетерогенні архітектури об’єднують центральні процесори (англ. central processing unit, CPU) та графічні процесори (англ. graphics processing unit, GPU) для виконання різних типів обчислень. CPU оптимізований для послідовних завдань, тоді як GPU – для паралельної обробки великих обсягів даних, наприклад, у сфері машинного навчання чи графічних обчислень.

Архітектура комп’ютера також визначає способи взаємодії між компонентами за допомогою системної шини. Існують різні стандарти шин, такі як PCIe для високошвидкісної передачі даних або USB для периферійних

пристроїв. Інші технології, наприклад, NVMe, дозволяють значно пришвидшити роботу дискової пам’яті завдяки використанню більш ефективних протоколів.

Не менш важливою є і адаптація архітектури до конкретних потреб. Наприклад, сервери часто використовують багатопроцесорні архітектури для обробки великих обсягів даних, тоді як мобільні пристрої орієнтовані на енергоефективність. ARM-архітектура, яка домінує в мобільних процесорах, забезпечує баланс між продуктивністю та низьким енергоспоживанням, що робить її ідеальною для смартфонів і планшетів.

Архітектура комп’ютера безпосередньо впливає на продуктивність програм. Наприклад, програмісти можуть використовувати знання про набір інструкцій або кеш-пам’ять для оптимізації своїх алгоритмів та реверс-інжинірингу, що особливо актуально в задачах моделювання, комп’ютерної графіки, обробки великих даних або виявлення та аналізу шкідливого програмного забезпечення.

У підсумку, можна сказати, що концепція архітектури комп’ютера є фундаментальною для розуміння роботи сучасних обчислювальних систем. Вона охоплює як базові принципи (модель фон Неймана, структура пам’яті), так і новітні підходи (гетерогенні архітектури, багатоядерність, віртуалізація тощо). Розуміння цих аспектів дозволяє створювати ефективні апаратні рішення, адаптовані до вимог конкретних завдань.

1.2. Стандарти та термінологія в комп'ютерній архітектурі

Основні стандарти та їх значення.

У сучасному світі інформаційних технологій стандартизація відіграє ключову роль у розробці, інтеграції та ефективному використанні апаратного й програмного забезпечення. Майбутні фахівці з IT повинні чітко розуміти, що стандарти є фундаментом, який забезпечує взаємодію систем, підвищує продуктивність і гарантує безпеку даних. Оскільки комп’ютерні системи складаються з великої кількості апаратних та програмних компонентів, які розробляються різними виробниками, то без визначення єдиних стандартів кожен виробник міг би використовувати власні підходи до обміну даними, побудови процесорів або розробки програмного забезпечення, що призвело б до несумісності пристроїв і програм. Наприклад, стандарт PCI Express (PCIe) гарантує, що будь-яка відеокарта, незалежно від виробника, може бути встановлена у відповідний роз’єм материнської плати. Аналогічно, стандарт USB (Universal Serial Bus) дозволяє підключати будь-які периферійні пристрої (флеш-накопичувачі, принтери, клавіатури) до комп’ютерів, незалежно від їхнього виробника.

Саме тому міжнародні організації, такі як ISO (International Organization for Standardization), IEC (International Electrotechnical Commission) та IEEE (Institute of Electrical and Electronics Engineers), розробляють і затверджують стандарти, які визначають правила побудови комп’ютерної архітектури, форматів

збереження даних, протоколів комунікації та принципів інформаційної безпеки. Наприклад, стандарт Unicode (ISO/IEC 10646) забезпечують коректне відображення тексту у багатомовних системах, дозволяючи комп’ютерам різних країн коректно обробляти символи національних алфавітів, а стандарт IEEE 754 визначає формат представлення чисел з плаваючою комою, що дозволяє програмам працювати однаково на різних типах процесорів, незалежно від виробника.

Так до основних стандартів, що забезпечують узгодженість у технологічній галузі можна віднести наступні:

ISO/IEC 2382:2015 – Information technology. Vocabulary – Інформаційні технології. Словник термінів[2]. Цей стандарт поділений на окремі частини (розділи), кожна з яких охоплює конкретну підгалузь IT. Цей стандарт – це «словник» IT-індустрії, який забезпечує точність і єдність термінології. Для студентів і фахівців він є основою для розуміння технічних текстів, написання документації та участі в глобальних проектах. Вивчення стандарту допомагає не лише знати визначення, а й розуміти логіку розвитку технологій.

ISO/IEC/IEEE 24765 – Systems and Software Engineering. Vocabulary – Інженерія систем і програмних засобів. Словник термінів [3]. Це міжнародний стандарт, який надає спільний словник термінів, застосовних до всіх аспектів системної та програмної інженерії. Його було розроблено для збору та стандартизації термінології, що використовується в цих галузях, з метою забезпечення єдиного розуміння та уникнення неоднозначностей у спілкуванні між професіоналами.основний наступник, який об’єднує термінологію IEEE, ISO та IEC.

ACPI (Advanced Configuration and Power Interface) – Вдосконалений інтерфейсу конфігурації та керування живленням [4]. Розроблений Intel, Microsoft, Toshiba, HP та Phoenix Technologies, цей стандарт визначає управління живленням та апаратним конфігуруванням систем, забезпечує взаємодію між операційною системою, апаратним забезпеченням та BIOS материнської плати.

PCI Express (PCIe) – Peripheral Component Interconnect Express – Експрес з’єднання периферійних компонентів [5]. Цей стандарт регламентує взаємодію високошвидкісних периферійних пристроїв, зокрема відеокарт, SSD, мережевих адаптерів тощо.

USB (Universal Serial Bus) – Універсальна послідовна шина [6]. Це стандарт інтерфейсу для підключення периферійних пристроїв до комп’ютерів та інших цифрових пристроїв. Його було розроблено у 1996 році консорціумом компаній Intel, Microsoft, IBM, Compaq, DEC, NEC та Nortel з метою уніфікації підключення пристроїв та спрощення взаємодії користувачів із технікою. USB замінив застарілі порти, такі як RS-232, PS/2, LPT та інші, забезпечивши швидку передачу даних, автоматичне виявлення пристроїв і можливість живлення підключених пристроїв.

IEEE 754 (ISO/IEC 60559) – Стандарт представлення чисел з плаваючою комою [7, 19]. Цей стандарт визначає формат представлення дійсних чисел у комп’ютерах, включаючи одинарну (32-біт), подвійну (64-біт) та розширену точність.

ISO/IEC 6093 – Information processing – Representation of numerical values in character strings for information interchange – Обробка інформації – Представлення числових значень у символьних рядках для обміну інформацією [8]. Цей стандарт надає три представлення числових значень у вигляді рядків символів, що зчитуються машиною, для використання в обміні між системами обробки даних, надає рекомендації щодо стандартів мов програмування розробки та продуктів програмування реалізації, так, що ці представлення можуть розпізнаються людьми. Стандарт застосовується лише до числових значень, що складаються зі скінченної кількості цифр з десятковим знаком або без нього.

ISO/IEC 10646 – Кодування символів [9]. Цей стандарт визначає «Універсальний набір кодованих символів» (Universal Coded Character Set, UCS). Він забезпечує уніфіковане кодування символів для представлення тексту більшості писемностей світу, а також технічних символів, пунктуації та інших елементів. Цей стандарт є основою для багатьох кодувань, включаючи популярне кодування UTF-8.

Таким чином, стандартизація є фундаментом для розвитку інформаційних технологій, дозволяючи різним системам взаємодіяти між собою, забезпечуючи сумісність пристроїв і програмного забезпечення, гарантуючи безпеку та стабільність роботи цифрових екосистем. Майбутні IT-спеціалісти, які володіють знаннями про ключові стандарти, матимуть значну перевагу на ринку праці, оскільки їхня експертиза дозволятиме створювати якісні, безпечні та масштабовані технологічні рішення. Розвиток сучасних технологій, зокрема штучного інтелекту, квантових обчислень і кіберфізичних систем, напряму залежить від узгоджених міжнародних стандартів, які дозволяють новітнім винаходам інтегруватися у вже існуючі цифрові платформи.

Основна термінологія та її використання.

Для правильного розуміння та застосування термінів в першу чергу потрібно ознайомитись з термінологією для точного опису понять, пов’язаних із обчислювальними машинами (ОМ) і комп’ютерами. Використання конкретних визначень термінів допомагає уникнути двозначностей, особливо у технічних галузях. Наприклад, слово «комп’ютер» у своєму найзагальнішому розумінні походить від слова «computer», що з англійської мови перекладається як «обчислювач», тому часто в більш старій літературі ще можна зустріти термін «обчислювальна машина».

На ранніх етапах розвитку таких обчислювальних машин, або комп’ютерів більшість завдань обмежувалися математичними розрахунками, зокрема

чисельними методами, аналізом даних, вирішенням систем рівнянь тощо, що і знайшло віддзеркалення в самому понятті «обчислювальна машина». Зберігання, обробка і комутація сигналів в ОМ в основному реалізується електронними схемами. Тому для ОМ довгий час використовувалася абревіатура ЕОМ (електронна обчислювальна машина). Цей термін з'явився у перших ЕОМ для їх відмінності від механічних і електромеханічних рахунково-обчислювальних пристроїв. У зв'язку з успіхами мікроелектроніки використання в ОМ складних електронних пристроїв, та званих «надвеликих інтегральних схем» (НВІС) термін ЕОМ став загальноприйнятим і звичним. Проте в даний час у літературі разом з терміном «обчислювальна машина», а слово «електронна» стосовно ОМ більше не несе істотної інформації. При цьому часто використовується термін «комп'ютер», цим віддається дань широкому розповсюдженню нового класу обчислювальної техніки – персональним ОМ. Однак слід пам’ятати, що комп’ютери охоплюють набагато ширший клас пристроїв, включно із суперкомп’ютерами, мейнфреймами, вбудованими системами тощо. Сучасні ОМ вийшли за рамки виключно обчислювальних завдань, оскільки вони можуть обробляти текстову, графічну, звукову інформацію, виконувати завдання штучного інтелекту та багато іншого. Крім того, ОМ містять не тільки пристрої зберігання і обробки інформації, але і пристрої периферії (або вводу і виводу). Якщо пристрої зберігання і обробки інформації реалізують на базі НВІС, то пристрої вводу і виводу разом з електронними компонентами містять електромеханічні, оптоелектронні та інші вузли. Тому в даний час переважно використовується термін «комп’ютер». Ми будемо, для зручності, використовувати обидва терміни.

В літературних джерелах, в залежності від автора та часу публікації можна знайти багато різних визначень навіть для таких термінів, як «обчислювальна машина» або «комп’ютер». Тому найбільш правильним буде використовувати визначення із міжнародного стандарту ISO/IEC 2382:2015 [2].

Комп’ютерна архітектура (computer architecture) – логічна структура та функціональні характеристики комп’ютера, що включає взаємозв’язки між його апаратними та програмними компонентами.

Комп’ютер (computer) – функціональний блок, який може виконувати значні обчислення, включаючи численні арифметичні та логічні операції без втручання людини. Комп’ютер може складатися з окремого блоку або кількох з’єднаних між собою блоків. В англійській мові в обробці інформації термін комп’ютер зазвичай відноситься до цифрового комп’ютера.

Система обробки даних, або **комп’ютерна система (data processing system, computer system, computing system) –** один або декілька комп'ютерів, периферійне обладнання та програмне забезпечення, що виконують обробку даних.

Мікрокомп’ютер (microcomputer) – цифровий комп'ютер, процесор якого складається з одного або кількох мікропроцесорів та включає засоби пам'яті та введення-виведення.

Центральне місце у структурі будь-якого комп’ютера займає процесор, тому почнемо з нього.

Мікропроцесор (microprocessor) – процесор, елементи якого були мініатюризовані в одну або кілька інтегральних схем.

Процесор (processor) – функціональний блок, який **Інтерпретує** та **Виконує інструкції.** Ключовими компонентами процесора є:

- блоку керування інструкціями (instruction control unit) – використовується для управління виконанням команд,

- арифметико-логічний блок (ALU) – використовується для виконання обчислювальних та логічних операцій.

Як вже зазначалось вище, процесор може бути також частиною більшого функціонального блоку – процесорного блоку.

Процесорний блок (Processing unit) – функціональний блок, який складається з одного або кількох процесорів і їх внутрішніх накопичувачів.

Відповідно: **центральний процесорний блок (Central processing unit (CPU))** – функціональний блок обробки, який є центральним для виконання функцій комп’ютера. Також декілька процесорів використовуються у багатопроцесорних системах, причому вони можуть бути організовані симетрично, як у симетричних багатопроцесорних системах **(SMP)**, де жоден із процесорів не є центральним.

Комп’ютер складається не лише з процесорів та пам’яті, а й з периферійного обладнання.

Периферійне обладнання (Peripheral equipment) – будь-який пристрій, яким керує комп’ютер і який може з ним взаємодіяти. Приклад: пристрої введення (клавіатура, миша); пристрої виведення (монітор, принтер); зовнішня пам’ять (жорсткі диски, флеш-накопичувачі) тощо.

Також розглянемо класифікацію комп’ютерів, що охоплює широкий спектр пристроїв:

Персональний комп’ютер (Personal computer (PC)) – мікрокомп'ютер, призначений переважно для автономного використання окремою особою

Головний комп’ютер / мейнфрейм (Mainframe) – комп'ютер, як правило, в комп'ютерному центрі, який має великі можливості та ресурси, і до якого можуть бути підключені інші комп'ютери, так щоб вони могли спільно використовувати його можливості та ресурси. Зазвичай ці комп’ютери розраховані на обслуговування численних користувачів і виконання складних обчислень.

Робоча станція (Workstation) – функціональний блок, який зазвичай має обчислювальні можливості спеціального призначення та включає орієнтовані на користувача блоки введення та блоки виведення. Приклад: програмований термінал, непрограмований термінал або автономний мікрокомп’ютер. Зазвичай

такі комп’ютери використовуються для наукових, інженерних чи графічних задач.

Термінал (terminal) – функціональний блок у системі чи комунікаційній мережі, у якому **Дані** можуть бути введені або отримані.

Дані (Data) – реінтерпретоване представлення інформації у формалізований спосіб, придатний для передачі, інтерпретації або обробки.

Інтерпретація (interpret) – аналіз, переклад та виконання кожного визначення (оператора) або кожної мовної конструкції у вихідній програмі перед обробкою наступного визначення (оператора).

Машинний код (machine code) – набір байтів для представлення допустимих різних машинних інструкцій конкретного комп'ютера.

Мова асемблера (assembly language) – машинно-орієнтована мова, яка забезпечує символічне найменування операцій та місць розташування, а також інші функції, такі як макроінструкції.

Програма (program) – синтаксичний блок, що відповідає правилам певної мови програмування і складається з **Оголошень, Визначень** та **Інструкцій**, необхідних для вирішення певної функції, завдання або проблеми.

Оголошення (declaration) – явна мовна конструкція, яка вводить один або більше ідентифікаторів у програму та визначає, як ці ідентифікатори мають інтерпретуватися.

Визначення, оператор (statement) – синтаксична одиниця з явним завершенням, що представляє оголошення або приписує одиницю роботи, яка включає ідентифікацію дій, які потрібно виконати, операндів (якщо такі є), які будуть використовуватися під час виконання цих дій, і розміщення будь-яких результатів.

Інструкція (instruction) – специфікація (визначення) операції та ідентифікація будь-яких пов’язаних операндів.

Операнд (operand) – сутність (об’єкт), над якою виконується операція.

Операція (operation) – чітко визначена дія, яка при застосуванні до будь-якої допустимої комбінації відомих сутностей створює нову сутність

Виконання (execution) – процес виконання інструкції або інструкцій комп'ютерної програми комп'ютером.

І наостанок розглянемо терміни з якими будемо працювати в першу чергу:

Цифра (numeral) – дискретне представлення числа. В різних системах числення одне й те саме значення числа може використовувати різні цифри: так «дванадцять» – у вигляді слова в українській мові, «twelve» – слово в англійській, число в десятковій системі числення – 12, в римській – XII, та 1100 – у двійковій системі.

Система числення (numeration system, number representation system) – будь-яка нотація для представлення чисел. Традиційно використовуються

цифрові системи числення – десяткова – людьми для повсякденної діяльності, а комп’ютерами – двійкова.

Ми розглянули лише невелику частину основних термінів, які знадобляться нам при вивчені дисципліни архітектура комп’ютера. Повністю ознайомитись із усіма визначеннями у вільному доступі можна на платформі онлайн-перегляду міжнародної організації зі стандартизації [2].

Висновки до розділу 1

Архітектура комп’ютера є фундаментальною основою, що визначає структуру та принципи функціонування обчислювальних систем. Вона охоплює як класичні моделі (фон-нейманівська, гарвардська), так і сучасні технології (багатоядерні процесори, гетерогенні архітектури, енергоефективні мобільні платформи). Знання архітектури комп’ютера має критичне значення для розробників програмного забезпечення, інженерів, системних адміністраторів та фахівців з кібербезпеки, оскільки дозволяє оптимізувати алгоритми, підвищувати продуктивність програм і розробляти новітні технологічні рішення. Сучасні обчислювальні системи потребують ефективного управління ресурсами – процесорами, пам’яттю, шинами та пристроями введення/виведення. Важливим аспектом є також адаптація архітектури під конкретні завдання: високопродуктивні сервери, енергоефективні мобільні пристрої чи спеціалізовані обчислювальні модулі для штучного інтелекту.

Стандартизація в інформаційних технологіях є фундаментальним інструментом, що забезпечує сумісність, безпеку та ефективність роботи апаратного та програмного забезпечення. Вона дозволяє різним пристроям і системам взаємодіяти між собою, усуваючи проблеми несумісності та полегшуючи інтеграцію новітніх технологій у глобальні цифрові екосистеми. Завдяки міжнародним стандартам, розробники, виробники та кінцеві користувачі можуть бути впевненими, що їхнє обладнання та програмне забезпечення працюватиме коректно незалежно від платформи чи виробника. Це особливо важливо у сфері кібербезпеки, мережевих технологій, збереження та обробки даних, де необхідні уніфіковані підходи до проєктування систем. Майбутні ІТ-фахівці, які розуміють стандарти та їхню роль у сучасних обчислювальних системах, отримують конкурентну перевагу, оскільки здатні створювати надійні, масштабовані та безпечні технологічні рішення. У міру розвитку штучного інтелекту, квантових обчислень та кіберфізичних систем стандартизація залишатиметься ключовим фактором, що сприятиме швидкому впровадженню нових технологій та їхньому гармонійному функціонуванню в сучасному цифровому світі.

Питання для самоконтролю

1. Що таке архітектура комп'ютера і чим вона відрізняється від організації комп'ютера?
2. Перелічіть основні компоненти архітектури фон Неймана та коротко опишіть призначення кожного з них.
3. У чому полягає проблема «вузького місця» (bottleneck) архітектури фон Неймана і як вона впливає на продуктивність системи?
4. Чим гарвардська архітектура відрізняється від архітектури фон Неймана? Назвіть переваги і недоліки кожної.
5. Що таке CPU і GPU? У чому принципова різниця між ними з точки зору типів задач, що вони виконують?
6. Які міжнародні організації займаються розробкою стандартів у галузі інформаційних технологій і які стандарти вони розробляють?
7. Для чого призначений стандарт IEEE 754 і які формати представлення чисел він визначає?
8. Що таке стандарт ISO/IEC 10646 і яке його практичне значення для сучасних комп'ютерних систем?
9. Поясніть різницю між термінами «комп'ютер», «мікрокомп'ютер» і «мікропроцесор» відповідно до стандарту ISO/IEC 2382:2015.
10. Що таке центральний процесорний блок (CPU) і які його ключові компоненти?
11. Чим персональний комп'ютер відрізняється від мейнфрейму і робочої станції?
12. Що таке периферійне обладнання? Наведіть приклади пристроїв введення і виведення.
13. Поясніть поняття «машинний код» і «мова асемблера». Як вони пов'язані між собою?
14. Що таке інструкція і операнд? Наведіть приклад.
15. Що таке система числення? Наведіть приклад представлення одного числа у різних системах числення.

РОЗДІЛ 2. СИСТЕМИ ЧИСЛЕННЯ ТА ПОДАННЯ ДАНИХ В ПАМ’ЯТІ КОМП’ЮТЕРА

2.1. Позиційні та непозиційні системи числення.

Система числення – це сукупність правил запису та оперування числами. Вона визначає, яким чином числа представляються та інтерпретуються в письмовій або цифровій формі. Системи числення бувають позиційними та непозиційними, і кожна з них має свої особливості, переваги та історичне значення.

Числові системи розвивалися разом із людством, починаючи з найпростіших позначок на кістках або каменях і доходячи до сучасних бінарних, десяткових і шістнадцяткових систем, що використовуються в комп’ютерній техніці.

Перші способи запису чисел були пов’язані з потребами людини рахувати предмети, вести облік запасів та проводити торгові операції. Археологічні знахідки свідчать, що ще 20-30 тисяч років тому первісні люди використовували насічки на кістках та дереві для позначення кількості об’єктів.

Найдавніші системи числення були непозиційними, тобто значення цифри не залежало від її місця у числі.

Однією з найдавніших систем числення вважається єгипетська (близько 3000 р. до н. е.). Єгиптяни користувалися десятковою системою числення, але непозиційною. Це означає, що символи мали значення незалежно від місця, де вони стояли (табл. 2.1):

1– вертикальна риска

10 – підкова або дуга

100 – спіраль, моток мотузки

1000 – лотос

10000 – вказівний палець

100000 – пуголовок (або жаба)

1000000 – людина з піднятими руками

Таблиця 2.1

Єгипетська система числення

1	10	100	1000	10 000	100 000	1 000 000
						

Числа складалися шляхом повторення потрібного символу, наприклад, 3 записували як три вертикальні риски, 40 – як чотири дуги, а 123 – одна спіраль (100), дві дуги (20) та три риски (3).

Подібну систему мали стародавні шумери (близько 3500 р. до н. е.), однак їхня система використовувала комбінацію десяткової та шістдесяткової основ. Арифметичні операції в такій системі були складними, оскільки не існувало компактного способу запису чисел.

У римській системі числення кожна цифра завжди має певне значення: I (1), V (5), X (10), L (50), C (100), D (500), M (1000), незалежно від того, де вона стоїть у записі числа (наприклад, "XVI" і "XIX"). Також використовувався віднімальний принцип (наприклад, IX = 9, XL = 40).

У грецькій та єврейській системах кожна буква алфавіту позначала число (α = 1, β = 2, γ = 3 і т. д.).

Головний недолік непозиційних систем – їхня складність у виконанні арифметичних операцій. Додавання, віднімання, множення та ділення в таких системах було громіздким і незручним.

Позиційні системи числення виникли пізніше та значно спростили математичні обчислення. Їхня ключова особливість полягає в тому, що значення цифри залежить від її позиції у записі числа.

Першою повноцінною позиційною системою була вавилонська система числення (близько 2000 р. до н. е.), яка базувалася на шістдесятковій основі (оснування 60). Вона використовувала символи для позначення одиниць та десятків, причому їхнє розташування впливало на значення числа.

Проте найважливішим кроком стало впровадження десяткової позиційної системи з використанням нуля. Це досягнення належить індійським математикам (близько 5-6 ст. н. е.), а потім було запозичене арабськими вченими та поширене в Європі через арабські трактати в середньовіччі.

Розглянемо представлення чисел у позиційній системі числення.

Позиційна система числення – це спосіб запису чисел, у якому значення цифри визначається не лише її самим значенням, а й позицією в числі. Кожна позиція відповідає певному ступеню основи системи числення.

Основа системи числення – кількість різних цифр або символів, які можуть бути використані для запису одного розряду числа. Так у десятковій системі (основа 10) дозволеними є цифри від 0 до 9. Виведемо загальну формулу позиційного числа. Число у позиційній системі числення з основою b (наприклад, $b = 10$ для десяткової системи) можна записати так:

$$N = d_k \cdot b^k + d_{k-1} \cdot b^{k-1} + \cdots + d_1 \cdot b^1 + d_0 \cdot b^0 + d_{-1} \cdot b^{-1} + d_{-2} \cdot b^{-2} + \cdots, \qquad (2.1)$$

де: d_i – цифра на позиції (розряді) i; b – основа системи числення; степінь i може бути як додатнім числом (для цілої частини), так і від’ємним (для дробової частини).

Приклад у десятковій системі (основа 10):

Розглянемо число $4725{,}38_{10}$ представлене у десятковій системі числення, про, що свідчить індекс 10. Його можна подати як суму чисел в кожному з розрядів, помножених на степені основи.

Варто звернути увагу, що номерація розрядів починається від коми: перший справа розряд (нульовий) є розрядом одиниць, або нульовим степенем основи, у нашому випадку $10^0 = 1$ і далі кожен розряд зліва збільшується в додатню сторону на порядок (степінь основи): перший розряд «десятків» – 10^1 (десять в першому степені), другий розряд «сотень» – 10^2 (десять в другому степені) і так далі. Відповідно дробові частки є діленням на відповідні степені основи, так перший розряд справа від коми є діленням на 10 (десяті частини), що може буде представлено як множення на основу у від’ємному степені, в нашому випадку 10^{-1}, другий розряд, відповідно, діленням на 100 (соті частини), або множенням на 10^{-2} і так далі.

Проаналізувавши число загалом бачимо, що цифра **4** стоїть на четвертій позиції зліва від коми, або тисяч, що дорівнює 10^3, так помноживши значення цифри в розряді на порядок $4 \cdot 10^3 = 4 \cdot 1000 = 4000$ отримаємо відповідну кількість тисяч. Далі цифра **7** знаходиться у третій позиції зліва від коми, або у сотнях (10^2), так $7 \cdot 10^2 = 7 \cdot 100 = 700$. Цифра **2** – на позиції десятків, $2 \cdot 10^1 = 2 \cdot 10 = 20$. Цифра **5** – на одиницях, $5 \cdot 10^0 = 5 \cdot 1 = 5$. Відповідно перейдемо до дробової частини , цифра **3** знаходиться на позиції десятих частин: $3 \cdot 10^{-1} = 3 \cdot 0{,}1 = 0{,}3$, а цифра **8** – на сотих, $8 \cdot 10^{-2} = 8 \cdot 0{,}01 = 0{,}08$. Відповідно розбиття за формулою (2.1) виглядає наступним чином:

$$4725{,}38 = 4 \cdot 10^3 + 7 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0 + 3 \cdot 10^{-1} + 8 \cdot 10^{-2} = \\ = 4000 + 700 + 20 + 5 + 0{,}3 + 0{,}08 = 4725{,}38$$

Отже, у позиційній системі числення кожна цифра має вагу, яка визначається основою та її позицією.

Як ми знаємо з початкових курсів математики, найменший крок в цілих числах є одиниця. Відповідно додавши або віднявши одиницю ми збільшуємо або зменшуємо число. Доходячи до максимального числа одному розряді ми отримуємо переповнення і маємо збільшити на одиницю старший розряд, а в молодший розряд записати найменше значення.

Наприклад, додавши до максимального однорозрядного числа 9 одиницю ми отримуємо переповнення і маємо поставити одиницю в розряд десятків, паралельно обнуливши розряд одиниць. Якщо в старшому розряді теж переповнення, то відповідно операція повторюється.

```
   9        1 9        9 9
 + 1      +   1      +   1
 ---      -----      -----
 1          1        1 1

 1 0        2 0      1 0 0
```

Десяткова система – лише один з прикладів, найпростіший для сприйняття за рахунок повсякденного використання. Проте, у комп’ютерних системах зазвичай використовуються системи числення з іншими основами: двійкова (основа 2), вісімкова (основа 8) та шістнадцяткова (основа 16), і тому розуміння принципу позиційності є базою для подальшого вивчення процесів обробки інформації, обміну та внутрішнього представлення даних у комп’ютері.

2.2. Двійкова, вісімкова та шістнадцяткова системи.

Як ми вже зазначали вище, числові системи – це методи запису чисел за допомогою символів та правил позиційного представлення. В сучасних інформаційних технологіях використовуються наступні декілька основних позиційних систем числення:

- Двійкова (англ. Binary, bin) – в основі системи числення лежить число 2;
- Вісімкова (англ. Octal, oct) – в основі системи числення лежить число 8;
- Десяткова (англ. Decimal, dec) – в основі системи числення лежить число 10;
- Шістнадцяткова (англ. Hexadecimal, hex) – в основі системи числення лежить число 16.

Кожна з них має свої особливості використання та застосовується в різних сферах науки і техніки, тому розглянемо кожну з них більш детально.

Двійкова система числення (основа 2)

Двійкова система використовує лише дві цифри: 0 і 1. Як вже зазначалось при переповненні молодшого розряду число переходить в старший розряд (табл.2.2)

Таблиця 2.2

Розрядності двійкових чисел

Двійкове число	0	1	10	11	100	101	110	111	1000	1001	1010
Розрядність двійкового числа	1	1	2	2	3	3	3	3	4	4	4
Десятковий аналог	0	1	2	3	4	5	6	7	8	9	10

Ось декілька прикладів запису чисел у двійковій системі:

$0_2 = 0_{10}$, $1_2 = 1_{10}$, $10_2 = 2_{10}$, $11_2 = 3_{10}$, $100_2 = 4_{10}$, $101_2 = 5_{10}$, $1000_2 = 8_{10}$, $1101_2 = 13_{10}$

NB! Саме від скорочення англійської назви двійкових чисел – «**Bi**nary digi**t**» походить назва найменшої одиниці інформації – **Bit, Біт**

Двійкова система числення є фундаментальною основою роботи сучасних обчислювальних пристроїв завдяки своїй простоті та надійності в технічній реалізації. розглянемо ключові аспекти того, як ця система забезпечує функціонування електроніки.

Апаратне забезпечення комп'ютера «розуміє» лише нулі та одиниці, це зумовлено тим, що комп'ютери побудовані на логічних електричних схемах, які в елементарному вигляді мають лише два стани: включено виключено. Так, для представлення дискретних змінних (0 та 1) використовуються безперервні фізичні величини, найчастіше – напруга в електричному ланцюзі, в на логічному рівні висока напруга зазвичай відповідає логічній одиниці (HIGH/TRUE), а низька напруга (рівень заземлення або GND) – логічному нулю (LOW/FALSE). Сучасні мікросхеми використовують КМОП-транзистори (n-МОП та p-МОП), які працюють як електрично керовані перемикачі, що пропускають або блокують проходження струму, формуючи таким чином стани 1 або 0.

Двійкова система ідеально поєднується з математичною логікою (Булевою логікою). Для логічних операцій використовуються логічні вентилі – це найпростіші цифрові схеми вентилі НЕ, І, АБО, XOR (виключне або) які приймають декілька двійкових сигналів на входи та видають новий сигнал на виході. Об’єднуючи ці прості блоки, інженери створюють складні модулі, такі як суматори для арифметичних операцій або блоки пам'яті. Наприклад, додавання двох двійкових чисел виконується за тими ж правилами, що й у десятковій системі, але набагато простіше в технічній реалізації. Тому будь-яка інформація в комп'ютері – від тексту до складних програм – зберігається у вигляді послідовностей бітів (двійкових розрядів).

Використання двійкового коду замість, наприклад, десяткового, має важливу перевагу – так набагато легше розрізнити два рівні напруги, ніж десять, це робить цифрові системи стійкими до електричного шуму, та дозволяє розробникам зосередитися на логіці роботи пристрою, ігноруючи складні фізичні процеси руху електронів у транзисторах. Таким чином, двійкова система числення є тією ланкою, що пов'язує фізичний світ електричних сигналів з інтелектуальним світом обчислень та програм.

У двійковому числі «вага» кожної позиції збільшується (так само, як і в десятковому – справа наліво) наступним чином: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 і т.д. Працюючи з двійковими числами, дуже корисно зберегти час запам'ятати значення ступенів двійки до 2^{16} (табл. 2.3).

Таблиця 2.3

Ступені двійки та діапазони значень

Ступінь 2^x	Кількість біт (знаків)	Діапазон значень (min ÷ max)
2^0	1 (x)	2 (0 ÷ 1)
2^1	2 (xx)	4 (0 ÷ 3)
2^2	3 (xxx)	8 (0 ÷ 7)
2^3	4 (xxxx)	16 (0 ÷ 15)
2^4	5 (x xxxx)	32 (0 ÷ 31)
2^5	6 (xx xxxx)	64 (0 ÷ 63)
2^6	7 (xxx xxxx)	128 (0 ÷ 127)
2^7	8 (xxxx xxxx)	256 (0 ÷ 255)
2^8	9 (x xxxx xxxx)	512 (0 ÷ 511)
2^9	10 (xx xxxx xxxx)	1 024 (0 ÷ 1023)
2^{10}	11 (xxx xxxx xxxx)	2 048 (0 ÷ 2047)
2^{11}	12 (xxxx xxxx xxxx)	4 096 (0 ÷ 4095)
2^{12}	13 (x xxxx xxxx xxxx)	8 192 (0 ÷ 8191)
2^{13}	14 (xx xxxx xxxx xxxx)	16 384 (0 ÷ 16383)
2^{14}	15 (xxx xxxx xxxx xxxx)	32 768 (0 ÷ 32767)
2^{15}	16 (xxxx xxxx xxxx xxxx)	65 536 (0 ÷ 65535)
2^{16}	17 (x xxxx xxxx xxxx xxxx)	131 072 (0 ÷ 131071)

NB! Крайній розряд зліва називається **найбільш значущім бітом (most significant bit)**, а крайній справа – **найменш значущим бітом (least significant bit)**.

Окрему увагу слід приділити дробовій частині двійкових чисел.

У десятковій системі числення кожен розряд праворуч від коми відповідає від'ємному степеню основи 10: перший розряд – $10^{-1} = 0.1$, другий – $10^{-2} = 0.01$, третій – $10^{-3} = 0.001$ і так далі. У двійковій системі діє той самий принцип, але основою є 2: перший розряд праворуч від коми відповідає $2^{-1} = 0.5$, другий – $2^{-2} = 0.25$, третій – $2^{-3} = 0.125$ і так далі.

Таким чином, двійковий дріб 0.101_2 розшифровується як:

$$0.1101_2 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

Якщо для переведення цілої частини числа використовується ділення на 2 з записом остач, то для дробової частини навпаки застосовується **метод множення на 2**: дробова частина послідовно множиться на 2, і ціла частина кожного результату (0 або 1) стає черговим бітом двійкового запису починаючи зліва від старшого біта, а дробова частина результату далі циклічно множиться на 2.

Приклад 1. Переведення 0.625_{10}:

Крок	**Дія**	**Результат**	**Ціла частина (біт)**
1	0.625 × 2	1.250	**1**
2	0.250 × 2	0.500	**0**
3	0.500 × 2	1.000	**1**

Якщо дробова частина стала рівною 0 – переведення завершено. Отриманий результат: $0.625_{10} = 0.101_2$

Перевірка: $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0 + 0.125 = 0.625$

Але більшість чисел не вдається так вдало порахувати.

Приклад 2. Переведення 0.1_{10}

Крок	**Дія**	**Результат**	**Ціла частина (біт)**
1	0.1 × 2	0.2	**0**
2	0.2 × 2	0.4	**0**
3	0.4 × 2	0.8	**0**
4	0.8 × 2	1.6	**1**
5	0.6 × 2	1.2	**1**
6	0.2 × 2	0.4	**0**
7	0.4 × 2	0.8	**0**
8	0.8 × 2	1.6	**1**
9	0.6 × 2	1.2	**1**
10	0.2 × 2	0.4	**0**
…	…	…	…

На кроці 6 отримуємо 0.2 – те саме значення, що й на кроці 2. Це означає, що послідовність бітів починає повторюватись циклічно і процес ніколи не завершиться.

Результат: 0.1_{10} = $0.0001100110011..._2 = 0.0(0011)_2$ (нескінченний періодичний дріб)

Період 0011 починається з 2-го біта і повторюється нескінченно. Оскільки пам'ять комп'ютера обмежена, і зберігається лише кінцева кількість біт – решта відкидається, що і спричиняє похибку представлення значення в комп’ютері. Так якщо спробувати перевести, обмежене 8 або 16 бітами двійкове число назад в десяткове, отримаємо трохи менше значення:

8 біт:

$$0.000\,1100_2 = 1 \times 2^{-4} + 1 \times 2^{-5} = 0.0625 + 0.03125 = 0.09375$$

16 біт:

$$0.000\,1100\,1100\,1100_2 = 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-8} + 1 \times 2^{-9} + 1 \times 2^{-12} + 1 \times 2^{-13} =$$
$$= 0{,}0625 + 0{,}03125 + 0{,}00390625 + 0{,}001953125 + 0{,}000244140625$$
$$+ 0{,}0001220703125 = 0{,}0999755859375$$

Як можна побачити, не всі дробові числа можна зберегти в пам’яті комп’ютера без похибок. Але для деяких операцій достатньо округлення до відносно точного числа. Більш детально представлення таких чисел в пам’яті комп’ютера ми розглянемо в розділі 2.5.

Вісімкова система числення (основа 8)

Вісімкова система використовує вісім цифр: 0, 1, 2, 3, 4, 5, 6, 7 – це, як можна помітити три двійкових розряди, що дозволяє легко переходити від двійкової системи (табл.2.4). Історично ця система числення популярною в системах, де довжина машинного слова була кратною трьом, і, відповідно, зараз вона також використовується в місцях де потрібно записати 3 двійкових значення, зокрема для скороченого запису рівнів доступу (читання, запис, виконання) в Unix-подібних операційних системах.

Приклад запису чисел у вісімковій системі:

$$7_8 = 7_{10};\ 10_8 = 8_{10};\ 17_8 = 15_{10};\ 24_8 = 20_{10}$$

Таблиця 2.4

Значення вісімкових чисел та аналогів у двійковій і десятковій системах числення

Десяткове значення	Вісімкове значення	Двійкове значення
0_{10}	0_8	000_2
1_{10}	1_8	001_2
2_{10}	2_8	010_2
3_{10}	3_8	011_2
4_{10}	4_8	100_2
5_{10}	5_8	101_2
6_{10}	6_8	110_2
7_{10}	7_8	111_2
8_{10}	10_8	$001\ 000_2$
10_{10}	12_8	$001\ 010_2$
15_{10}	17_8	$001\ 111_2$
16_{10}	20_8	$010\ 000_2$
23_{10}	27_8	$010\ 111_2$
24_{10}	30_8	$011\ 000_2$
63_{10}	77_8	$111\ 111_2$
64_{10}	100_8	$001\ 000\ 000_2$

Шістнадцяткова система числення (основа 16)

Шістнадцяткова система відповідно до назви використовує 16 символів:

- цифри десяткової системи 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- та латинські літери A, B, C, D, E, F, де A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

Шістнадцяткова система числення застосовується для скороченого запису двійкових чисел при програмуванні, адресації пам’яті, в кольорових кодах тощо. Кожен шістнадцятковий розряд відповідає 4 двійковим цифрам (табл.2.5).

Приклад запису чисел у шістнадцятковій системі

$A_{16} = 10_{10} = 1010_2$; $F_{16} = 15_{10} = 1111_2$; $1F_{16} = 31_{10} = 0001\ 1111_2$;

$FF_{16} = 255_{10} = 1111\ 1111$; $100_{16} = 256_{10} = 0001\ 0000\ 0000_2$

NB! Два шістнадцяткових числа складають 8 біт, або 1 байт інформації. Відповідно найстарша і наймолодші пара шістнадцяткових чисел називаються найбільш та найменш значущими байтами.

Таблиця 2.5

Значення шістнадцяткових чисел та аналогів у двійковій і десятковій системах числення

Десяткове значення	Шістнадцяткове значення	Двійкове значення
0_{10}	0_{16}	0000_2
1_{10}	1_{16}	0001_2
2_{10}	2_{16}	0010_2
3_{10}	3_{16}	0011_2
4_{10}	4_{16}	0100_2
5_{10}	5_{16}	0101_2
6_{10}	6_{16}	0110_2
7_{10}	7_{16}	0111_2
8_{10}	8_{16}	1000_2
9_{10}	9_{16}	1001_2
10_{10}	A_{16}	1010_2
11_{10}	B_{16}	1011_2
12_{10}	C_{16}	1100_2
13_{10}	D_{16}	1101_2
14_{10}	E_{16}	1110_2
15_{10}	F_{16}	1111_2
16_{10}	10_{16}	$0001\ 0000_2$
31_{10}	$1F_{16}$	$0001\ 1111_2$
32_{10}	20_{16}	$0010\ 0000_2$
255_{10}	FF_{16}	$1111\ 1111_2$
256_{10}	100_{16}	$0001\ 0000\ 0000_2$

2.3. Переведення чисел між системами числення

Переведення десяткового числа в інші системи числення.

Переведення десяткового числа в інші системи числення виконується методом послідовного цілочисельного ділення на основу потрібної системи числення з виділенням остачі (в програмуванні для виділення остачі часто використовується знак %), доки результат не буде дорівнювати нулю, і в подальшому записі остачі від ділення у зворотному порядку. Розглянемо приклади переведення десяткового числа 2025_{10} в двійкову (приклад 1) вісімкову (приклад 2) та шістнадцяткову (приклад 3) системи числення:

Приклад 1а. Переведення десяткового числа в двійкове:

1. $2025 \div 2 = 1012$, остача 1 – записуємо в наймолодший, 0 (нульовий) розряд.
2. $2012 \div 2 = 506$, остача 0 – записуємо в 1 розряд;
3. $506 \div 2 = 253$, остача 0 – записуємо в 2 розряд;
4. $253 \div 2 = 126$, остача 1 – записуємо в 3 розряд;
5. $126 \div 2 = 63$, остача 0 – записуємо в 4 розряд;
6. $63 \div 2 = 31$, остача 1 – записуємо в 5 розряд;
7. $31 \div 2 = 15$, остача 1 – записуємо в 6 розряд;
8. $15 \div 2 = 7$, остача 1 – записуємо в 7 розряд;
9. $7 \div 2 = 3$, остача 1 – записуємо в 8 розряд;
10. $3 \div 2 = 1$, остача 1 – записуємо в 9 розряд;
11. $1 \div 2 = 0$, остача 1 – результат 0 – ділення завершено, записуємо в старший, 10 розряд.

Результат (записуємо остачі у зворотному порядку): 11111101001_2.

NB! Для зручності при записі двійкове число рекомендується ділити на тетради – по 4 цифри починаючи з наймолодшого розряду (зліва) та заповнюючи порожні старші розряди нулями: $0111\ 1110\ 1001_2$

Приклад 2а. Переведення десяткового числа в вісімкове:

1. $2025 \div 8 = 253$, остача 1 – записуємо в 0 (молодший) розряд;
2. $253 \div 8 = 31$, остача 5 – записуємо в 1 розряд;
3. $31 \div 8 = 3$, остача 7 – записуємо в 2 розряд;
4. $3 \div 8 = 0$, остача 3 – записуємо в 3 (старший) розряд.

   Результат: 3751_8

Приклад 3а. Переведення десяткового числа в шістнадцяткове:

1. $2025 \div 16 = 126$, остача 9 – записуємо в 0 (молодший) розряд;
2. $126 \div 16 = 7$, остача 14 – записуємо відповідне десятковому 14 значення E в 1 розряд;
3. $7 \div 16 = 0$, остача 7 – записуємо в 2 (старший) розряд.
   Результат: $7E9_{16}$.

NB! Аналогічно до двійкових чисел – шістнадцяткові при записі рекомендується ділити попарно, заповнюючи старші розряди нулями: $07\ E9_{16}$

Переведення двійкового, вісімкового та шістнадцяткового чисел в десяткове.

Переведення інших систем числення в десяткову виконується методом розкладання числа на розряди, множення значення в розряді на відповідну степінь основи з подальшою сумою отриманих значень. Для наочності розглянемо обернені до попередніх приклади.

Приклад 1б. Переведення двійкового числа 11111101001_2в десяткове відбувається наступним чином:

1. Визначаємо розряди зліва на право, які використовуємо як степені двійки.

Значення	1	1	1	1	1	1	0	1	0	0	1
Розряд	10	9	8	7	6	5	4	3	2	1	0
Степінь 2	1024	512	256	128	64	32	16	8	4	2	1

2. Перемножуємо значення на степінь і додаємо:

$$11111101001_2 =$$
$$= 1 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$$
$$= 1 \cdot 1024 + 1 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 =$$
$$= 2025_{10}$$

Приклад 1в. Для двійкових чисел з невеликою кількістю нулів може використовуватись метод віднімання. Для цього береться розряд на один старше найбільш значущого біта мінус одиниця і від цього числа віднімаються лише ті розряди, у яких у вихідному числі стоїть нуль.

Враховуючи табличку з попереднього прикладу розраховуємо:

$$11111101001_2 = (2^{11} - 1) - 1 \cdot 2^4 - 1 \cdot 2^2 - 1 \cdot 2^1 = (2048 - 1) - 16 - 4 - 2 = 2025_{10}$$

Приклад 2б. Перетворимо вісімкове число 3751_8 у десяткове:

1. Визначаємо розряди зліва на право, які використовуємо як степені вісімки

Значення	3	7	5	1
Розряд	3	2	1	0
Степінь 8	512	64	8	1

2. Перемножуємо значення на степінь і додаємо:

$$3751_8 = 3 \cdot 8^3 + 7 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 = 3 \cdot 512 + 7 \cdot 64 + 5 \cdot 8 + 1 \cdot 1 = 2025_{10}$$

Приклад 3б: Перетворимо шістнадцяткове число $7E9_{16}$ у десяткове:

1. Визначаємо розряди зліва на право, які використовуємо як степені 16:

Значення в шістнадцятковій СЧ	7	E	9
Значення в десятковій СЧ	7	14	9
Розряд	2	1	0
Степінь 16	256	16	1

2. Перемножуємо значення на степінь і додаємо:

$$7E9_{16} = 7 \cdot 16^2 + E \cdot 16^1 + 9 \cdot 16^0 = 7 \cdot 256 + \cdot 14 \cdot 16 + \cdot 9 \cdot 1 = 2025_{10}$$

Переведення між двійковою, вісімковою та шістнадцятковою системами числення.

Для переведення між двійковою, вісімковою та шістнадцятковою системами числення використовується через двійкову систему методом групування двійкових цифр по 3 та 4 розряди для вісімкової та шістнадцяткової систем відповідно. Групування двійкових цифр виконується справа наліво.

Приклад 4. Для переведення з двійкової в вісімкову і назад ділимо по групами по 3 цифри (так звані «тріади»), порожні старші розряди, за необхідності, заповнюємо нулями:

$11111101001_2 \rightarrow \underline{0}11\ 111\ 101\ 001$

$011_2 = 3_8$; $111_2 = 7_8$; $101_2 = 5_8$; $001_2 = 1_8$

Результат: 3751_8.

І зворотній розрахунок для числа 547_8:

$5_8 = 101_2$; $4_8 = 100_2$; $7_8 = 111_2$.

Результат: 101100111_2

Приклад 5. Для переведення з двійкової в шістнадцяткову і назад ділимо по групами по 4 цифри («тетради»), порожні старші розряди, за необхідності, заповнюємо нулями:

$11111101001_2 \rightarrow \underline{0}111\ 1110\ 1001$
$0111_2 = 7_{16}$; $1110_2 = 14_{16}$; $1001_2 = 9_{16}$.

Результат: $7E9_{16}$.
І зворотній розрахунок для числа 547_{16}:

$5_{16} = 0101_2$; $4_{16} = 0100_2$; $7_{16} = 0111_2$.
Результат: 10101000111_2

Приклад 6. Для переведення з вісімкової в шістнадцяткову і назад виконується проміжне переведення в двійкову систему, при цьому порожні старші розряди можуть доповнюватись нулями. Використаємо зворотні розрахунки з попередніх прикладів.

- Переведемо вісімкове число 547_8 в шістнадцяткову систему числення:

$547_8 = 101\ 100\ 111_2 \rightarrow \underline{000}1\ 0110\ 0111_2 = 167_{16}$

- Переведемо шістнадцяткове число 547_{16} в вісімкову систему числення

$547_{16} = 0101\ 0100\ 0111_2 \rightarrow 010\ 101\ 000\ 111_2 = 2507_8$

Ми розглянули основні принципи формування чисел в різних системах числення та перетворення (переведення) чисел між цими системами, розуміння яких є важливим у всіх сферах інформаційних технологій, зокрема в інженерних розрахунках для програмування, цифрової електроніки, обчислювальної техніки та мережевих налаштувань.

Подання даних в пам’яті комп’ютера

В усіх комп’ютерах, заснованих на базі цифрової електроніки дані на найнижчому рівні зберігаються у вигляді бітів із значеннями одиниця або нуль. Найменша адресована одиниця інформації називається **Байт (byte)** – зазвичай це октет, який містить 8 біт. Одиниця інформації, яка оброблюється інструкціями машинного коду, називається **Словом (Word)** розмірністю в 16, 32 або 64 біти в залежності від архітектурних особливостей процесора: розрядності регістрів процесора та/або розрядності шини даних.

Дані в залежності від організації комп'ютера, можуть бути доступні в одному або кількох різних розмірах, однак один із доступних розмірів майже завжди дорівнює машинному слову, а інші розміри, в більшості випадків будуть кратними слову або частинами розміру машинного слова, менші розміри зазвичай використовуються лише для економного використання пам'яті: під час завантаження в процесор їхні значення, як правило, потрапляють до більшої комірки – регістру розміром зі слово. Так машинні слова можуть використовуватися для таких даних:

- Числа з фіксованою комою, переважно цілі числа;

- Числа з рухомою комою, зазвичай мають розмір одного слова або кратні йому.

- Адреси – комірки для зберігання адрес пам'яті;

- Регістри процесора – проєктуються з розміром, що відповідає типу даних, які вони зберігають: цілим числам, числам з рухомою комою або адресам (у багатьох комп'ютерних архітектурах використовуються регістри загального призначення, здатні зберігати дані в кількох форматах);

- Передача даних між пам'яттю і процесором – обсяг переданих даних найчастіше дорівнює одному слову.

- Одиниці адресного розподілу – в будь-якій архітектурі послідовні значення адрес майже завжди позначають послідовні одиниці пам'яті, ця одиниця і є одиницею адресного розподілу.

- Інструкції машинного коду (команди) зазвичай мають розмір, що дорівнює слову архітектури або є його часткою, оскільки інструкції та дані зазвичай використовують одну підсистему пам'яті (слід звернути увагу, що в Гарвардській архітектурі розміри слів для інструкцій і даних можуть не збігатися, оскільки інструкції та дані зберігаються в різних видах пам'яті).

Розглянемо більш детально представлення цілих чисел в комп’ютері (приклади подання чисел для 4-бітного слова наведено на рисунку 2.1). Так числа можуть бути представлені в комп’ютері у вигляді **прямого коду** (без знакове та знакове представлення), **оберненого коду** (доповнення до одного або англ. ones' complement) та **доповняльного коду** (також, в різних джерелах, визначається як доповнюючий код або доповнення до двох, англ. two's complement):

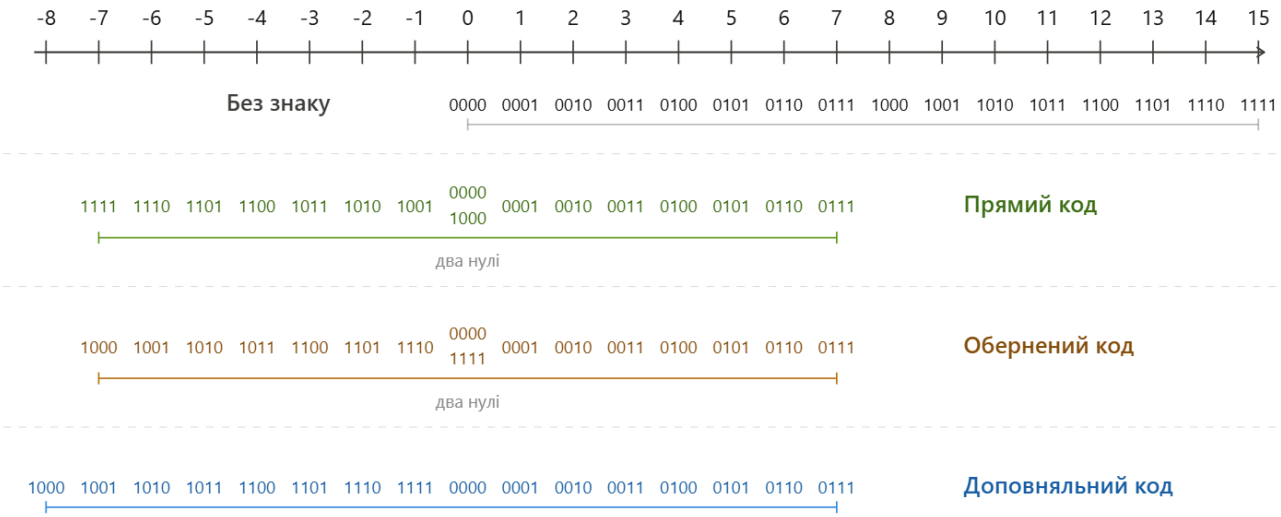

Рисунок 2.1 – Прямий без знаку та зі знаком, обернений та доповняльний коди
[схему побудовано з використанням LLM Claude Sonnet 4.6, Anthropic, https://claude.ai]

- **прямий код** є одним із методів представлення двійкових чисел із нерухомою комою в комп'ютерній арифметиці, що застосовується переважно для кодування невід'ємних величин. Для переведення числа в прямий код достатньо перевести число в двійкову систему числення, а потім заповнити вільні зліва розряди нулями (на рис.2.1 рядок «Без знаку»). У більш традиційних випадках, коли прямий код поширюється на знакові числа, тобто на числа, які допускають як додатні, так і від'ємні значення, числова частина доповнюється окремим знаковим розрядом або для знаку використовується найбільш значущий розряд, що визначає наявність знаку мінус (на рис. 2.1 рядок «Прямий код»). Слід звернути увагу, що переведення у вигляді прямого коду, хоч і є найбільш зручним для людського сприйняття, має декілька суттєвих недоліків, таких як наявність двох, додатнього та від’ємного, нулів (+0 та –0) та потреба в розробці спеціальних механізмів віднімання в процесорі, оскільки немає можливості побітово віднімати числа. Приклади для 4-розрядних чисел в прямому коді:

$$5_{10} = 0101_2, -5_{10} = 1101_2,$$
$$0_{10} = 0000_2 = 1000_2 = -0_{10},$$
$$2_{10} + 3_{10} = 0010_2 + 0011_2 = 0101_2 = 5_{10},$$
$$3_{10} - 2_{10} = 0011_2 - 0010_2 = 0001_2 = 1_{10},$$
$$3_{10} + (-2_{10}) = 0011_2 + 1010_2 = 0001_2 = 1_{10},$$
$$(-3_{10}) + 2_{10} = 1011_2 + 0010_2 = 1001_2 = -1_{10}.$$

Як можна побачити, працювати з прямим кодом для чисел з різним знаком може бути не дуже зручно, особливо для процесора.

- **обернений код** – один із способів подання двійкових цілих чисел зі знаком. Щоб отримати обернений код із десяткового числа, його спершу переводять у

двійкову форму без урахування знака. Якщо число від’ємне, всі біти інвертуються (0 замінюється на 1, а 1 – на 0). З математичної точки зору, віднімання в оберненому коді проводиться як додавання від’ємного значення початкового числа, але з поправкою на перенесення переповнення в молодший біт. Отже, цей метод подання інформації дозволяє замінити віднімання на операцію додавання, але всеодно має складності: вимагає додаткової операції циклічного переносу (при переповненні) зі старшого біту в молодший, та не вирішує проблему двох нулів (рис.2.1 «Обернений код»). Ті ж приклади 4-розрядних чисел в оберненому коді:

$$5_{10} = 0101_2, -5_{10} = 1010_2,$$
$$0_{10} = 0000_2 = 1111_2 = -0_{10}$$

Окремо більш детально розглянемо приклади арифметичних операцій:

3 + (-2):

```
  0011    (+3)
+ 1101    (-2 в оберненому коді знаходиться наступним чином:
           беремо +2 = 0010, інвертуємо всі біти → 1101)
──────
 10000    → перенос із старшого біта
+    1    → додаємо його до найменшого
──────
  0001    =  (+1) результат операції
```

Приклад -3 + 2:

```
  1100    (-3 в оберненому коді: 3₁₀=0011, і після інверсії отримуємо 1100
+ 0010    (+2)
──────
  1110    (-1, переносу зі старшого розряду немає).
```

Можна побачити, що наявність переносу ускладнює реалізацію арифметичних розрахунків в оберненому коді.

- **доповняльний код** – це найпоширеніший спосіб представлення від’ємних чисел у комп’ютерах. Якщо число від’ємне, всі біти інвертуються і до числа додається 1. Застосування доповняльного коду дозволяє замінити операцію віднімання на додавання як для знакових, так і для беззнакових чисел без зайвих додаткових операцій, що значно спрощує архітектуру комп’ютера та зменшує кількість необхідних команд, оскільки процесору не потрібно розрізняти знакове та беззнакове числа, не потрібно робити перенос біта переповнення (він просто

відкидається), а також вирішується проблему з двома нулями (оскільки 0 = 0000, а -0 = 1111 + 1 = 1̶ 0000 (перенос відкидається). Також доповняльний код має в своєму діапазоні додаткове від’ємне число, яке не має додатної пари. Так, набір значень для 4 бітного слова наведено на рис.2.1 – «Доповняльний код».

Якщо розглянути попередні приклади

$$5_{10} = 0101_2, -5_{10} = 1010_2 + 1_2 = 1011_2,$$

-5 в доповняльному коді знаходиться наступним чином: беремо $+5_{10} = 0101_2$, інвертуємо всі біти → 1010_2, додаємо 1 → 1011_2. Спробуємо інвертувати 0:

$$0_{10} = 0000_2 = 1111_2 + 1_2 = 0000_2,$$

як можна побачити, додавання 1 вирішує проблему від’ємного нуля, замість нього стає -1:

$$1111_2 = 0000_2 + 1_2 = 0001_2 = 1_{10},$$
$$8_{10} = 1000_2 \rightarrow 0111_2 + 1_2 = 1000_2 = -8_{10}$$

Можна помітити, що поява від’ємного нуля уникається, і, більш того, з’являється додаткове від’ємне число, у якого не має додатної відповідності.

Тепер розв’яжемо арифметичні приклади:

3 + (-2):

```
  0011    (+3)
+ 1110    (-2 в доповняльному коді)
------
 10001    перенос із старшого біта
 ↑        відкидається
------
  0001    = (+1) результат операції.
```

(-3) + 2:

```
  1101    (-3)
+ 0010)
------
  1111    = (-1) результат операції
```

Також наочною може бути схема циклічних переносів, відображена на рисунку 2.2, де діапазони 4-бітних цілих чисел –чорне кільце; беззнакові числа – біле кільце; обернений код – помаранчеве; доповняльний код – бірюзові; а також відображено червоною стрілкою ефект додавання 4 до довільного значення.

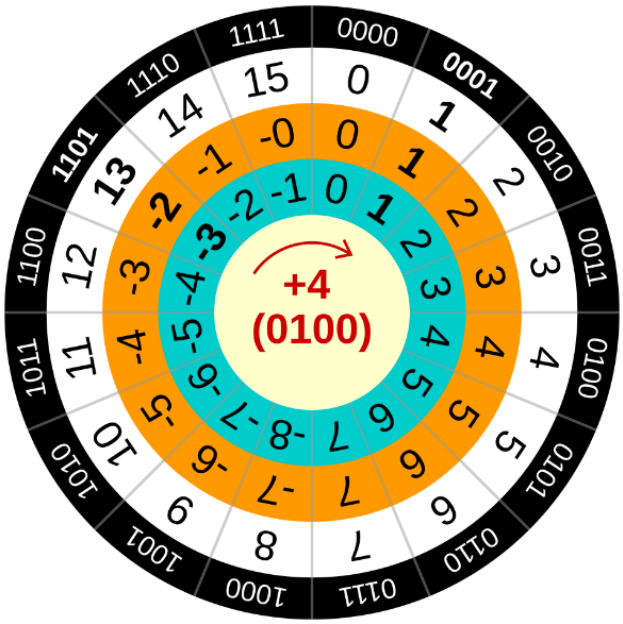

Рисунок 2.2 – Циклічне зображення діапазонів 4-бітних цілих чисел
[https://en.wikipedia.org/wiki/File:Twos_vs_ones_complement_circle.svg]

2.4. Арифметичні операції у двійковій системі.

Арифметичні операції в двійковій системі числення є основою роботи всіх комп’ютерних пристроїв. Оскільки комп’ютери працюють лише з двома станами (0 і 1), виконання операцій додавання, віднімання, множення та ділення здійснюється відповідно до правил двійкової арифметики.

Додавання та віднімання у двійковій системі

Додавання та віднімання в двійковій системі подібні до десяткового, але тут використовуються лише дві цифри: 0 і 1.

Основні правила додавання:

№	A	B	Сума	Перенесення
1.	0	0	0	0
2.	0	1	1	0
3.	1	0	1	0
4.	1	1	0(2)	1
–	**A**	**B**	**Перенесення вхідне**	**Сума**	**Перенесення**
5.	1	1	1	1	1

Варто звернути увагу, що при додаванні двох одиниць отримуємо двійку ($1+1 = 2 = 10_2$), але двійки в двійковій системі немає, тому в поточному розряді записується нуль, а одиниця переноситься в старший розряд. Окремим рядком винесено і правило, коли в одному розряді додаються три одиниці ($1+1+1 = 3 = 11_2$) – сума 1 і перенесення 1. Це максимально можливе значення в одному розряді і основа роботи **повного суматора** в процесорі.

Найзручніше такі операції виконувати в стовпчик. Розглянемо приклади:

Приклад 1. Без перенесення

```
  0011    (3)
+ 0100    (4)
──────
  0111    = 7 – Результат операції
```

Кожен розряд додається незалежно, перенесень немає.

Приклад 2. З перенесенням

```
  0111    (7)
+ 0001    (1)
──────
  0111
+ 0001
──────
  1000    = 8 – Результат операції
```

Проходимо порозрядно справа наліво:

- позиція 0: 1+1 = 0, перенесення 1
- позиція 1: 1+0+1(перенесення) = 0, перенесення 1
- позиція 2: 1+0+1(перенесення) = 0, перенесення 1
- позиція 3: 0+0+1(перенесення) = 1

Цей приклад відображає класичний ланцюжок перенесень – кожен розряд «передає» далі одиничку перенесення. Саме такі випадки найдовше виконуються в апаратних суматорах.

При відніманні в двійковій системі замість позичання «десятка» позичають «двійку». Основні правила віднімання:

№	A	B	Різниця	Позичання
1.	0	0	0	0
2.	0	1	1	1
3.	1	0	1	0
4.	1	1	0	0
–	**A**	**B**	**Позичання вхідне**	**Різниця**	**Позичання**
5.	0	0	1	1	1
6.	0	1	1	0	1
7.	1	0	1	0	0
8.	1	1	1	1	1

При відніманні одиниці від нуля (рядок 2 та 5) позичаємо 1 у старшого розряду, що дорівнює $10_2-1=1$, і фіксуємо позичання.

Іншими важливими прикладами є перенесення позичання коли від нуля (рядок 6) або одиниці (рядок 8) віднімається дві одиниці: одиниця другого числа і позичання з молодшого розряду. В результаті отримуємо, відповідно, $10_2–1–1=0$ і $11_2–1–1=1$, вихідне значення розряду зберігається, а позичання передається далі старшому розряду.

Розглянемо також декілька прикладів:

Приклад 1. Без позичання

```
  0111    (7)
- 0100    (4)
-------
  0011    = 3 – Результат операції
```

Кожен розряд віднімається без позичання із старшого розряду.

Приклад 2. З позичанням

```
  1010    (10)
- 0011    (3)
-------
  1010
- 0011
-------
  0111    = 7 – Результат операції
```

Проходимо порозрядно справа наліво:

- позиція 0: 0−1 = 1, позичання 1 (позичаємо з позиції 1)
- позиція 1: 1−1−1(позич.) = 1, позичання 1 (позичаємо з позиції 2)
- позиція 2: 0−0−1(позич.) = 1, позичання 1 (позичаємо з позиції 3)
- позиція 3: 1−0−1(позич.) = 0, позичання 0

Варто звернути увагу, що додавання та віднімання розглядалось для беззнакових чисел. У реальних комп’ютерних системах старший (лівий) розряд, як правило, відведено під знак числа, як ми розглядали вище – 0 означає, що число додатнє, а 1 – від'ємне. Так для, наприклад, 4-розрядного об’єму даних діапазон значень звужується і під модуль числа залишається лише 3 розряди. Виділення найбільш значущого розряду під знак необхідно враховувати при інтерпретації результатів арифметичних операцій, наприклад останній перенос або запозичення може бути виконано в неіснуючий біт, значення якого у випадку обмеження пам’яті, наприклад, одним байтом (8 біт) може бути втрачено.

Переповнення виникає коли результат арифметичної операції виходить за межі діапазону, який можна представити в розрядній сітці – перенесення або запозичення із старшого біту або змінюють знак коли це не потрібно, або взагалі не зберігають результат для беззнакових чисел.

Приклад 1. Переповнення при додаванні

```
  0101    (+5)
+ 0110    (+6)
──────
  1011    = −5 – Результат операції – неправильний
```

Обидва доданки додатні, але результат 1011 у знаковій 4-бітній сітці інтерпретується як від'ємне число – старший біт став 1.

Приклад 2. Переповнення при відніманні

```
  1010    (−6)
- 0101    (+5)
──────
  0101    = +5 – Результат операції – неправильний
```

Від'ємне число мінус додатне дало додатній результат. Математично ($-6-5=-11$), але -11 також виходить за межі діапазону.

Приклад 3. Втрата даних при переносі (для беззнакових чисел)

```
  1111    (15)
+ 0001    (1)
──────
 10000    = 16, але 5-й біт не вміщується у 4-бітну сітку
  0000    = 0 – Результат операції – неправильний
```

Результат математично правильний – 16 = 10000_2, але оскільки розрядна сітка має лише 4 біти, п'ятий біт просто **губиться**, і в пам'яті залишається 0000.

NB! Варто не плутати **переповнення** та **втрату старшого біта при перенесенні**:

Переповнення (overflow) – результат виходить за межі діапазону знакової сітки, результат отримує неправильний знак.

Перенос (carry) – результат не вміщується в розрядну сітку взагалі, старший біт просто губиться. Це актуально для беззнакових чисел:

Додавання та віднімання в доповняльному коді

Головна перевага арифметичних операцій в доповняльному коді полягає в тому, що виконується лише звичайне двійкове додавання, без жодних додаткових правил. Віднімання, по-суті, зводиться до додавання – від'ємник перетворюється на від'ємне число у доповняльному коді і далі виконується операція звичайного додавання. Саме тому застосування доповняльного коду є таким зручним для реалізації – процесору не потрібен окремий пристрій для віднімання.

Приклад 1. Додатнє + додатнє

```
  0011    (+3)
+ 0010    (+2)
──────
  0101    = +5 – Результат операції
```

Приклад 2. Додатнє + від'ємне

```
  0101    (+5)
+ 1110    (−2 в доповняльному коді)
──────
 10011    → перенос відкидаємо
  0011    = +3  −результат операції правильний
```

Як отримали −2: +2 = 0010 → інвертуємо → 1101 → +1 → 1110.

Приклад 3. Від'ємне + від'ємне

```
  1101    (−3)
+ 1110    (−2)
──────
 11011    → перенос відкидаємо
  1011    = −5  – результат операції правильний
```

Перевірка: 1011 – старший біт 1, отже від'ємне. Інвертуємо, отримуємо 0100, додаємо 1, отримуємо $0101_2 = 5_{10}$, а, отже, наше число −5.

Ключове правило арифметики в доповняльному коді: **перенос із старшого розряду завжди відкидається** – це не помилка, а нормальна робота доповняльного коду. Помилкою є лише переповнення, яке ми вже розглянули вище.

Інші недоліки обмеженої розрядності.

1. Втрата точності (недостатня дискретизація)

У комп’ютерах числа зберігаються в обмеженій кількості бітів, що може спричинити похибки округлення. Навіть коли дробові числа представлені у вигляді з плаваючою комою – часто вони не можуть бути представлені точно. Наприклад, десяткове число 0.1_{10} у двійковій системі є нескінченним періодичним дробом: $0.0001100110011...._2$. Оскільки пам’ять обмежена, зберігається лише частина цього числа, що спричиняє невелику похибку в обчисленнях.

Ракетна система Patriot, 1991.

Американська зенітна ракетна система Patriot під час війни в Перській затоці використовувала внутрішній лічильник часу з кроком 0.1 секунди. Як ви вже описали, 0.1 не представляється точно у двійковій системі – похибка становила близько 0.000000095 секунди на крок. За 100 годин безперервної роботи накопичена похибка склала **0.34 секунди**, що для ракети, що летить зі швидкістю 1.5 км/с, означало похибку визначення позиції у **500 метрів**. Система не змогла перехопити іракську ракету Scud, яка вбила 28 американських солдатів.

2. Підтікання (underflow)

Підтікання виникає, коли число настільки мале або велике, що не може бути представлене у відведеній розрядності. Наприклад, якщо ми використовуємо 4-

бітове представлення з фіксованою точкою, то найменше ненульове число може бути $0001_2 = 1_{10}$. Якщо результат обчислень буде меншим за 1, він округлюється до 0, що спричиняє втрату інформації.

Ракета Aerian-5, 1996.

Ракета Аріан-5 ціною 7 мільярдів доларів, запущена 4 червня 1996 року, відхилилася від курсу та зруйнувалася через 37 секунд після запуску. Відмова була викликана тим, що у бортовому комп'ютері при конвертуванні 64-розрядного числа з плаваючою комою в 16-розрядне ціле зі знаком сталося переповнення, після якого комп'ютер завис. Програмне забезпечення Аріан-5 було ретельно протестовано, проте на ракеті Аріан-4. Однак нова ракета мала двигуни з більш високими швидкісними параметрами, які, будучи переданими до бортового комп'ютера, і викликали переповнення регістрів.

3. Нестабільність обчислень через накопичення помилок

Якщо виконувати багато арифметичних операцій з числами, які мають похибки округлення, ці помилки накопичуються. У наукових обчисленнях це може призводити до значних відхилень у кінцевому результаті.

Нафтова платформа Sleipner A, 1991.

Норвезька нафтова платформа вартістю **700 мільйонів доларів** затонула під час випробувань через помилки в розрахунках міцності конструкції. Інженери використовували метод скінченних елементів для моделювання напружень у бетонних стінках платформи. Через недостатню точність дискретизації моделі похибки в розрахунках напружень накопичились і реальне навантаження на стінки виявилось на **47% більшим** за розраховане. Під час занурення стінка не витримала тиску води і платформа затонула. Розслідування показало, що проблема була саме в накопиченні чисельних похибок при багаторазових ітераційних обчисленнях.

Методи боротьби з переповненням і втратою точності

- Контроль переповнення – процесори мають спеціальні флаги, які сигналізують про переповнення або перенос із старшого розряду і дозволяють викликати процедуру обробки виключення та опрацювати переповнення.
- Збільшення розрядності – використання чисел більшої розрядності, зокрема 64 або 128-бітних замість 32-бітних дозволяє зменшити ймовірність переповнення, а також у багатьох мовах програмування є спеціальні типи для обчислень без обмежень за розрядністю (наприклад, `BigInt` у JavaScript).

- Використання чисел з плаваючою комою – для представлення чисел з великою динамікою діапазону (наприклад, у фінансових або наукових розрахунках) використовуються формати представлення чисел із плаваючою комою, описані в стандарті IEEE 754.

Множення у двійковій системі

Множення виконується аналогічно до множення у стовпчик у десятковій системі, але значно простіше, оскільки кожен біт множника може бути лише 0 або 1, то таблиця множення виглядає набагато простіше:

№	A (множене)	B (множник)	Результат (добуток)
1.	0	0	0
2.	0	1	0
3.	1	0	0
4.	1	1	1

А кожен частковий добуток – це або 0, або копія множеного зсунута на відповідну кількість розрядів. Множення двох 4-бітних чисел може дати результат до 8 біт.

Загальне правило: добуток двох n-бітних чисел може потребувати до 2n біт для точного представлення. Саме тому в процесорах результат множення зазвичай зберігається в подвійному регістрі.

Приклад 1. Додатнє × додатнє

```
    0011   (+3)
×   0010   (+2)
 ————————
    0000
+   0011
+  0000
+ 0000
 ————————
 00000110  = +6 – Результат операції, старші біти (до 8 біт заповнюються нулями)
```

Для від’ємних чисел в доповняльному коді множення в стовпчик не може бути застосоване, тому, якщо рахувати вручну – можна інвертувати в додатні беззнакові числа, виконати операцію множення і перевести результат в доповняльний код, але в комп’ютерній техніці для збільшення ефективності застосовується **алгоритм Бута**.

Алгоритм Бута (Booth's multiplication algorithm) – це метод множення чисел у доповняльному коді, який базується на аналізі пар сусідніх бітів множника. Його головна ідея – використовуючи властивість двійкових серій одиниць, замінити операції зі знаком та послідовним додаванням на більш ефективні операції. Замість того, щоб додавати множене для кожної одиниці в множнику, алгоритм Бута «бачить» групи одиниць. Алгоритм виконає одне віднімання на початку серії одиниць і одне додавання в кінці, що економить обчислювальні ресурси при роботі з довгими послідовностями бітів.

Розглянемо більш детально алгоритм:

Нехай $\boldsymbol{m}$ та $\boldsymbol{r}$ – це множене та множник, а $\boldsymbol{x}$ та $\boldsymbol{y}$ – відповідна кількість бітів у $\boldsymbol{m}$ та $\boldsymbol{r}$.

1. Підготовка.

Для роботи алгоритму створюється спільна робоча область (конвеєр), що складається з трьох частин:

- **Акумулятор (A):** Розмірність $\boldsymbol{x}$ біт. Спочатку заповнений нулями. Тут накопичується результат.
- **Множник (P):** Сюди записується число $\boldsymbol{r}$, на яке ми множимо.
- **Додатковий біт ($\boldsymbol{P_{-1}}$):** Додатковий уявний біт праворуч від множника, який на початку дорівнює нулю.

2. Аналіз пар бітів.

На кожному кроці алгоритм аналізує на два наймолодші (крайні праві) біти: останній біт множника ($\boldsymbol{P_0}$) та додатковий біт ($\boldsymbol{P_{-1}}$). Залежно від їхньої комбінації виконується одна з дій:

- **01**: "Кінець серії одиниць". До акумулятора ($\boldsymbol{A}$) додається множене $\boldsymbol{m}$: $A = A + m$.
- **10**: "Початок серії одиниць". Від акумулятора ($\boldsymbol{A}$) віднімається множене $\boldsymbol{m}$ (або додається його від’ємне значення в доповняльному коді): $A = A - m = A + (-m)$.
- **00** або **11**: "Всередині серії". Ніяких математичних дій з акумулятором не проводиться.

3. Арифметичний зсув (ASR)

Після аналізу та можливих операцій додавання/віднімання вся конструкція конвеєру арифметично зсувається вправо на один біт.

4. Циклічність.

Пункти 2 та 3 циклічно повторюються. Кількість ітерацій алгоритму точно дорівнює кількості бітів $\boldsymbol{y}$ в множнику $\boldsymbol{r}$.

5. Отримання результату.

Після завершення останнього зсуву уявний біт $\boldsymbol{P_{-1}}$ відкидається, а результат зчитується безпосередньо з регістрів $\boldsymbol{A}$ та $\boldsymbol{P}$ як єдине ціле число. Із зазначеного можна зробити висновок, що для множення використовується подвійна кількість

бітів і результат відповідно буде зберігатись в ділянці пам’яті (регістрі) подвоєної розрядності.

Зсув – це операція переміщення всіх бітів числа на одну або кілька позицій ліворуч або праворуч у розрядній сітці. Зсуви широко використовуються в комп'ютерній арифметиці, оскільки зсув на n позицій ліворуч рівнозначний множенню на 2^n, а зсув праворуч – діленню на 2^n.

Логічний зсув – усі біти зсуваються на задану кількість позицій, а позиції, що звільняються завжди заповнюються нулями незалежно від знаку числа. Біти, що виходять за межі розрядної сітки відкидаються.

Приклад логічного зсуву праворуч на 1 позицію:

`1110 → 0111` (старший біт замінюється нулем)

Логічний зсув застосовується для беззнакових чисел.

Арифметичний зсув – при зсуві праворуч позиція, що звільняється зліва заповнюється **знаковим бітом** (тобто зберігає знак числа), а не нулем. При зсуві ліворуч поведінка така сама як у логічного зсуву – заповнення нулями справа.

Приклад арифметичного зсуву праворуч на 1 позицію:

`1110 → 1111` (старший біт 1 зберігається)

`0110 → 0011` (старший біт 0 зберігається)

Арифметичний зсув застосовується для знакових чисел у доповняльному коді, оскільки зберігає знаковий біт (знак) при діленні на 2. Саме цей вид зсуву використовується в алгоритмі Бута.

Приклад: $4 \times (-5)$ в 4-бітному вигляді

Виконуємо підготовку до розрахунків:

Вихідні дані:

множене $m = 4,$ множник $r = -5;$ розмірності множників $x = y = 4$ біт,

Значення: $m = 0100,\ -m = \overline{0100} + 1 = 1011 + 1 = 1100,$

$r = \overline{0101} + 1 = 1011$

A (акумулятор) $= 0000,$

P (множник) $= 1011,$

P_{-1}(додатковий біт праворуч від P) $= 0$

Послідовність кроків:

Крок	Стан / Операція	A	P	P_{-1}	Коментар
0	Початковий стан	`0000`	`101`**`1`**	**`0`**	Аналізуємо P_0 та P_{-1}

1	$A = A - m$	**1100**	1011	0	Оскільки $P_0P_{-1} = 10$ віднімаємо m від A (додаємо $-m = 1100$)
	Зсув вправо (ASR)	**1**110	010**1**	**1**	Виконуємо арифметичний зсув (із збереженням знакового біту 1 на початку) та знову аналізуємо P_0 та P_{-1}
2	Зсув вправо (ASR)	**1**111	001**0**	**1**	Оскільки $P_0P_{-1} = 11$, виконується лише арифметичний зсув вправо знову аналізуємо P_0 та P_{-1}
3	$A = A + m$	**0011**	0010	1	Оскільки $P_0P_{-1} = 01$ додаємо m (0100) до A (1111+0100=0011) переповнення відкидається
	Зсув вправо (ASR)	**0**001	100**1**	**0**	Виконуємо арифметичний зсув (із збереженням знакового біту 0 на початку) та знову аналізуємо P_0 та P_{-1}
4	$A = A - m$	**1101**	1001	0	Оскільки $P_0P_{-1} = 10$ віднімаємо m від A (додаємо $-m = 1100$) (0001+1100=1101)
	Зсув вправо (ASR)	**1110**	**1100**	1	Фінальний арифметичний зсув (із збереженням знакового біту 1 на початку). Відкидаємо додатковий уявний біт P_{-1}. Об’єднане значення A та P і є результатом множення.

Перевіряємо результат:

1. Число починається з 1, отже воно від’ємне.

2. Переводимо з доповняльного коду в прямий (аналогічно – інвертуємо біти і додаємо одиницю:

$$\overline{1110\ 1100} + 1 = 0001\ 0011 + 1 = 0001\ 0100_2 = 20_{10}$$

3. Додаємо знак і отримуємо -20, що відповідає завданню $4 \times (-5) = -20$

Результат правильний.

Ділення у двійковій системі

Ділення – найскладніша для реалізації арифметична операція. На відміну від множення, де алгоритм Бута дає елегантне рішення, ділення не має настільки універсального підходу.

Загальний принцип ділення у двійковій системі виконується аналогічно до ділення у стовпчик у десятковій системі: ділене порівнюється з дільником, визначається черговий біт частки, віднімається частковий добуток, залишок зсувається і процес повторюється. Для ділення чисел зі знаком, система має запам’ятати знаки і виконати ділення абсолютних значень в прямому коді і потім, якщо знаки діленого та дільника відрізняються привести частку до формі зі знаком (наприклад в доповнений код)

Приклад: $10111_2 \div 10_2$ $(23_{10} \div 2_{10})$

```
 10111 / 10 = 1
-10
──────
  01   / 10 = 0 – не ділиться, переходимо до наступного розряду
──────
   11  / 10 = 1
  -10
──────
    11 / 10 = 1
   -10
──────
     1 – остача від ділення
```

Результат: 1011_2 або 11_{10} і остача 1

Існує два основних підходи до ділення:

1. Ділення з відновленням остачі

На кожному кроці від поточного залишку віднімається дільник:

• якщо залишок ≥ 0 записуємо 1 в біт частки і зберігаємо залишок;

• якщо залишок < 0 записуємо в біт частки 0, а залишок **відновлюється** (до залишку назад додається дільник).

2. Ділення без відновлення остачі

Це більш оптимізований варіант у порівнянні із попереднім: якщо на попередньому кроці залишок був від'ємним, замість відновлення на цьому кроці – дільник **додається на наступному кроці** замість віднімання. Це, в гіршому випадку вдвічі скорочує кількість арифметичних операцій.

Особливості апаратної реалізації

На відміну від додавання і множення, ділення погано піддається розпаралелюванню та оптимізації, оскільки кожен крок залежить від результату

попереднього і виконання операції ділення може займати на порядок більшу кількість тактів процесора.

Саме тому якщо дільник **відомий заздалегідь** часто намагаються замінити множенням на обернену величину (1/b), обраховану один раз і збережену у вигляді змінної: замість a ÷ b обчислюють b1=1/b, і далі по ходу роботи програми розраховуються a × b1. Проте ця схема має ще одне обмеження – 1/ b – не має бути нескінченим дробом.

Ділення на степінь двійки

Окремий випадок – ділення на 2^n, яке виконується миттєво через арифметичний зсув праворуч на n позицій, без жодних ітерацій:

6 ÷ 2 = 3: 0110 зсувається праворуч на 1 розряд і отримаємо результат 0011

8 ÷ 4 = 2: 1000 зсувається праворуч на 2 розряди і отримаємо результат 0010

2.5. Подання дробових чисел з фіксованою та плаваючою комою.

У комп’ютерних обчисленнях числа можуть бути представлені у двох основних форматах: фіксованої коми (fixed-point) та плаваючої коми (floating-point). Представлення чисел в форматі з фіксованою комою підходить для задач, що не потребують високої точності та можуть застосовуватись у вузькому діапазоні значень, тоді як числа з плаваючою комою дозволяють працювати з широким діапазоном чисел, включаючи дуже великі та дуже малі значення з високою точністю, але вони займають досить багато місця (від 16 до 128 біт) та вимагають додаткових обрахунків для їх представлення.

Більшість десяткових дробів не можуть бути представлені точно у двійковій системі числення і при збереженні в пам'яті обрізаються до фіксованої кількості біт. Це означає, що **будь-яке число з плаваючою комою є лише наближенням до істинного значення**.

При виконанні багатьох послідовних операцій ці похибки наближення можуть накопичуватись і суттєво вплинути на результати обчислень. Саме тому у фінансових розрахунках, де важлива швидкодія та є критичною абсолютна точність обрахунків (наприклад до копійок), як правило, уникають чисел з плаваючою комою і використовують цілочисельну арифметику з фіксованою комою або спеціалізовані бібліотеки довільної точності. А от в системах які вимагають великої точності, таких як моделювання наукових експериментів або розрахунок траєкторій руху космічних тіл, використовуються переважно числа з плаваючою комою.

Числа з фіксованою комою (fixed-point numbers) – це спосіб представлення дробових чисел у комп’ютерних системах, де позиція десяткової (або двійкової) коми залишається незмінною. На відміну від чисел з плаваючою комою (floating-

point), тут усі числа мають однакову точність і фіксовану кількість бітів для дробової частини.

У системах із фіксованою комою число записується у двійковій формі, а кома (роздільник між цілою та дробовою частиною) умовно знаходиться у певному місці, яке задається програмістом або апаратною архітектурою.

Наприклад, якщо використовуємо 8-бітне число у форматі 4.4 (4 біти на цілу частину, 4 – на дробову), то число 1010.1100_2 буде інтерпретоване як: "1010" = 10 (ціла частина), а ".1100" = 0.75 (дробова частина):

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (0 \times 2^{-4}) = \\ = 8 + 2 + 0.5 + 0.25 = 10.75$$

Отже, 1010.1100_2 у десятковій системі дорівнює 10.75_{10}.

Застосування чисел з фіксованою комою має свої переваги: швидші обчислення, ніж із плаваючою комою; просте представлення, що не вимагає складних алгоритмів округлення; та передбачувана точність, оскільки розрядність дробової частини фіксована.

В цілому такий формат використовується у аудіообробці та цифрових сигналах; у банківських і фінансових системах; у вбудовані системах та мікроконтролерах з обмеженими ресурсами.

Представлення чисел з плаваючою комою, та правила виконання операцій із ними визначаються стандартом IEEE 754 (ISO/IEC 60559), що забезпечує універсальність та точність їх використання в комп’ютерних системах.

Стандарт IEEE 754 був запроваджений у 1985 році й визначає формат збереження та виконання арифметичних операцій з числами з плаваючою комою, на сьогодні остання версія стандарту IEEE 754-2019 [7], його зміст ідентично відображено в міжнародному стандарті ISO/IEC 60559:2020 [18].

Число у форматі IEEE 754 представлено за такою формулою:

$$(-1)^S \times M \times 2^E,$$

де: S (Sign bit, 1 біт) – знак числа (0 – додатне, 1 – від’ємне);

M (Mantissa, мантиса) – нормалізоване дробове число, що містить значущі цифри;

E (Exponent, експонента) – показник ступеня, який масштабує число.

Мантиса для забезпечення як великих, так і малих чисел має зміщення на середину свого діапазону, що обчислюється за формулою $2^{n-1} - 1$, де n – кількість бітів експоненти. Основні формати чисел з плаваючою комою, що визначаються стандартом наведено в таблиці 2.6.

Таблиця 2.6

Основні формати які визначає стандарт IEEE 754

Формат	**Розрядність**	**Знак** (біт)	**Експонента** (біт)	**Зміщення експоненти**	**Мантиса** (біт)[В]	**Приблизний діапазон** значень	**Точність** (дес. знаків)
Половинна точність	16 біт	1	5	F	10	$\sim\pm10^{\pm4}$	~3
Одинарна точність	32 біти	1	8	7F	23	$\sim\pm10^{\pm38}$	~7
Розширена одинарна[А]	43+ біт	1	11+	3FF+	31+	–	–
Подвійна точність	64 біти	1	11	3FF	52	$\sim\pm10^{\pm308}$	~15
Розширена подвійна[Б]	80 біт	1	15	3FFF	64[Г]	$\sim\pm10^{\pm4932}$	~18
Чотирна точність[Б]	128 біт	1	15	3FFF	112	$\sim\pm10^{\pm4932}$	~34

Для таблиці 2.6 є декілька важливих приміток:

А. Розширена одинарна точність – стандарт визначає лише мінімальні вимоги (43+ біти), конкретна реалізація залежить від виробника, тому діапазон і точність не фіксовані.

Б. Розширена подвійна і чотирна точності мають однакове зміщення експоненти, але різну мантису, тому діапазон допустимих значень однаковий, але точність їх апроксимації – різна.

В. Для двійкових форматів мантиса містить лише біти дробової частини, без врахування прихованого цілого біта "1",

Г. В 80-бітному форматі цілий біт мантиси є явним і входить в ці 64 біти мантиси.

Приклад. Розглянемо запис числа -12.625 у форматі одинарної точності, 32 біти.

Крок 1. Переведення числа у двійкову систему

1. Ціла частина: $12_{10} = 1100_2$.
2. Дробова частина: $0.625_{10} = 0.5 + 0.125 = 2^{-1} + 2^{-3} = 0.101_2$.
3. Разом: $12.625_{10} = 1100.101_2$.

Крок 2. Нормалізація

Число потрібно привести до вигляду $1.M \times 2^E$:

$$1100.101_2 = 1.100101_2 \times 2^3.$$

- Мантиса (M): 100101 (дробова частина. Ціла частина – уходить в прихований розряд).
- Порядок (E): 3 (степінь двійки нормалізованого числа).

Крок 3. Розрахунок полів формату

1. Знак (1 біт): Число від’ємне, тому знак = 1.
2. Експонента (8 біт): До розрахованого порядку числа додаємо зміщення 127_{10} (шістнадцяткове $7F_{16}$, або двійкове $0111\ 1111_2$).

$$3_{10} + 127_{10} = 3_{16} + 7F_{16} = 11_2 + 0111\ 1111_2 = 130_{10} = 82_{16} = 1000\ 0010_2.$$

3. Мантиса (23 біти): Беремо дробову частину після коми та дописуємо нулі справа до 23 знаків.

1001 0100 0000 0000 0000 000.

Крок 4. Фінальний результат

Об'єднуємо всі частини в один 32-бітний рядок:

Знак (S)	Експонента (E)	Мантиса (M)
1	10000010	10010100000000000000000

Далі отриманий набір двійкових символів для зручності відображення переводиться в шістнадцяткову форму (розділимо по 4 біти):

1100 0001 0100 1010 0000 0000 0000 0000,

що відповідає шістнадцятковому запису: C14A0000.

2.6. Типи даних

Враховуючи, що всі дані в сучасних комп’ютерах зберігаються у вигляді одиниць та нулів важливо розуміти і те, як процесор розпізнає дані які в цих наборах містяться. Так для розпізнавання команд у процесора є вказівник команд, який в процесі компіляції або інтерпретації коду програми вказує, що конкретна ділянка пам’яті є командою, а для інтерпретації даних – є визначені кожною мовою програмування типи даних.

Тип даних – це набір або група значень даних, зазвичай заданих набором можливих значень, набором дозволених операцій над цими значеннями та представленням цих значень як машинних типів. Специфіка типу даних обмежує можливі значення, які може приймати вираз, такий як змінна або функція. Іншими словами він повідомляє **компілятору** або **інтерпретатору**, тобто комп’ютеру, як програміст має намір використовувати дані.

Машинні типи даних - це спеціальні типи даних, які використовуються для зберігання і обробки числових значень, таких як цілі числа або числа з плаваючою комою (апроксимовані значення дробових чисел), з точністю до певного байтового розміру. Вони зазвичай визначені платформою або мовою програмування і можуть відрізнятися від мови до мови та від архітектури до архітектури.

Машинні типи даних надають можливість точного контролю над використаною пам'яттю і оптимізацією, але вони також можуть вимагати уважного управління діапазонами і відсутністю переповнень (наприклад, переповнення цілих чисел). Розмір машинних типів даних може різнитися в залежності від платформи, тому важливо бути обережним із сумісністю при перенесенні програм між різними системами.

Основними машинними типами даних є наступні:

- Цілочисельні типи даних – **Integers;**
- Тип даних з плаваючою комою – **Floating-Point Numbers;**
- Логічний тип даних – **Booleans**;
- Символьний тип даних – **Characters;**
- Байтові типи даних – **Bytes**;
- Перелічувальний тип даних – **Enumerated**;
- Інші складні типи даних.

Цілі числа (Integers):

Цілочисельні типи даних використовуються для зберігання цілих чисел, тобто чисел без десяткової частини. Ці типи даних дозволяють працювати з цілими числами різного розміру та діапазону. Основні цілочисельні типи даних включають наступні:

- int: Це стандартний цілочисельний тип даних із знаком, який зазвичай використовується для зберігання цілих чисел. Розмір типу int може варіюватися на різних платформах, але зазвичай він має розмір 32 біти або 64 біти.
- short int (short): Це цілочисельний тип даних із знаком меншого розміру, ніж int. Розмір short зазвичай 16 біт.
- long int (long): Це цілочисельний тип даних із знаком більшого розміру, ніж int. Розмір long зазвичай 32 біти або 64 біти, залежно від платформи.

- long long int (long long): Це цілочисельний тип даних із знаком ще більшого розміру, який може зберігати дуже великі цілі числа. Розмір long long зазвичай 64 біти.
- unsigned int: Це цілочисельний тип даних без знаку, який використовується для зберігання невід'ємних цілих чисел. Він також може мати розмір 32 біти або 64 біти, залежно від платформи.

Як цілочисельні беззнакові типи даних можуть також використовуватись byte (від 0 до 255) та word (від 0 до 65535), які ми розглянемо пізніше.

Цілочисельні типи даних важливі для багатьох аспектів програмування, таких як арифметичні операції, зберігання даних, робота з масивами та інші операції.

Числа з плаваючою комою (Floating-Point Numbers):

Зазвичай регулюються стандартом IEEE 754

Типи даних з плаваючою комою використовуються для зберігання чисел які можуть мати дробову частину. Основні типи даних з плаваючою комою включають наступні:

- float: Тип даних float використовується для зберігання чисел з рухомою комою з одинарною точністю. Він зазвичай використовує 32 біти для подання чисел і забезпечує відносну точність для чисел з десятковою комою.
- double: Тип даних double використовується для зберігання чисел з рухомою комою з подвійною точністю. Він зазвичай використовує 64 біти для подання чисел і забезпечує більшу точність і діапазон для чисел з десятковою комою порівняно з типом float.
- long double: Тип даних long double використовується для зберігання чисел з рухомою комою зі збільшеною точністю. Розмір long double може варіюватися від платформи до платформи, і він зазвичай подає числа з ще більшою точністю і діапазоном порівняно з типом double.

Типи даних з плаваючою комою корисні для обробки дійсних чисел, таких як грошові суми, вага, температура, різноманітні коефіцієнти, тощо. Важливо враховувати, що в обчисленнях з числами з рухомою комою можуть виникати питання щодо точності, і деякі операції можуть вести до втрати точності або неправильних результатів. Тому важливо обирати тип даних з плаваючою комою залежно від конкретних вимог задачі.

Булеві значення (Booleans):

Логічні типи даних, також відомі як булеві типи даних, використовуються для зберігання булевих значень, які можуть бути лише двох видів: True (правда) або False (неправда). Ці типи даних широко використовуються для умовних

виразів, логічних операцій та в управлінні потоком програми. Основні логічні типи даних включають наступні:

- bool (boolean): Тип bool є основним логічним типом даних і використовується для зберігання значень True або False. Цей тип часто використовується в умовних виразах та логічних операціях.
- bit: У деяких низькорівневих мовах програмування, таких як C або C++, можливо використовувати бітовий тип для зберігання одного біта і вручну обробляти значення True або False, використовуючи бітові операції у вигляді одиничи або нуля.

Логічні типи даних є важливим елементом умовних конструкцій (наприклад, умовних операторів if-else), циклах, логічних операціях (наприклад, логічні "І", "АБО" і "НЕ") та інших аспектах програмування. Вони дозволяють програмам приймати рішення на основі умов і керувати потоком виконання відповідно до цих умов.

Символи (Characters):

Символьні типи даних використовуються для зберігання символів, таких як літери, цифри, розділові знаки та інші видимі та недруковані символи. Символьні типи даних дозволяють програмам працювати з текстовими рядками та символьними данними. Основні символьні типи даних включають:

- char: Тип char (від англ. "character") використовується для зберігання одного символу. Найчастіше це використовується для літер, але також може бути використано для зберігання інших символів, таких як цифри або розділові знаки.
- wchar_t: Тип wchar_t (від "wide character") використовується для зберігання широких символів, які можуть включати символи з різних кодувань, такі як UTF-16 або UTF-32. Цей тип даних використовується в мовах програмування, які підтримують роботу з юнікодом.
- char16_t та char32_t: Ці типи даних використовуються для роботи зі широкими символами у форматах UTF-16 і UTF-32 відповідно. Вони дозволяють зберігати символи з більшою точністю і підтримують роботу з Unicode.
- string: Це тип даних, який представляє рядок символів. У більшості мов програмування рядок є складним типом даних, який містить послідовність char або wchar_t. Рядковий тип даних дозволяє зберігати та обробляти текстову інформацію.
- char[] або char*: В деяких мовах програмування, таких як C або C++, рядок може бути представлений масивом символів char або вказівником на перший символ рядка.

Символьні типи даних дозволяють обробляти текстову інформацію, виконувати операції порівняння символів, шукати та замінювати символи у рядках і виконувати інші операції, пов'язані з роботою з текстом. Важливо

враховувати кодування символів при роботі з char, wchar_t, char16_t і char32_t, оскільки це впливає на те, як символи представляються в пам'яті і оброблюються програмою.

Однобайтний символьний тип може використовуватись для представлення символів з набору ASCII та інших восьмирозрядних кодувань, тоді як для представлення символів з набору Unicode потрібно щонайменше 2 байти.

Байти (Bytes):

Байтові типи даних використовуються для зберігання і обробки даних на рівні байтів, які є основними одиницями інформації в комп'ютерних системах. Ці типи даних дозволяють точно контролювати використану пам'ять і представлення даних у вигляді послідовностей байтів. Основні байтові типи даних включають наступні:

- byte (байт): Тип byte використовується для зберігання одного байта даних, який може приймати значення від 0 до 255. Цей тип даних дуже корисний для роботи з бінарними даними, наприклад, для зберігання зображень, аудіофайлів, або інших форматів даних, де важлива точність.
- sbyte (знаковий байт): Тип sbyte також використовується для зберігання одного байта, але він дозволяє приймати значення від -128 до 127, оскільки цей тип є зі знаком.
- byte[] (масив байтів): Масив байтів використовується для зберігання послідовності байтів. Цей тип даних дуже корисний для роботи з бінарними даними, буферами, файлами та іншими операціями, які вимагають маніпуляції байтами.

Байтові типи даних зазвичай використовуються для оптимізації роботи з пам'яттю та для низькорівневого програмування, такого як робота з мережевими пакетами, зберігання бінарних даних і робота з пристроями вводу/виводу. Важливо враховувати, що розмір байта може різнитися в залежності від архітектури комп'ютера, і вони можуть бути зі знакоми або без знаку, залежно від мови програмування.

Перелічувальні типи даних:

Перелічувальний тип даних, або перелік - тип даних, що описується шляхом перелічення всіх можливих значень (кожне з яких позначається власним ідентифікатором), які можуть приймати об'єкти даного типу. Імена нумераторів зазвичай є ідентифікаторами, які поводяться як константи в мові. Змінній, яка була оголошена як така, що має перерахований тип, можна призначити будь-який з перелічувачів як значення. Іншими словами, перерахований тип має значення, які відрізняються одне від одного, і які можна порівнювати та призначати, але не визначені програмістом як такі, що мають якесь конкретне представлення в пам’яті комп’ютера; компілятори та інтерпретатори можуть представляти їх довільно.

Перелічуваний тип визначається як набір ідентифікаторів, з погляду мови програмування відіграють таку ж роль, як і звичайні іменовані константи, але пов'язані з цим типом. Класичний опис типу-перерахування у мові Паскаль виглядає так:

type Cardsuit = (clubs, diamonds, hearts, spades);

Тут проводиться оголошення типу даних Cardsuit (карткова масть), значеннями якого може бути будь-яка з чотирьох перерахованих констант. Змінна типу Cardsuit може приймати одне з значень clubs, diamonds, hearts, spades, допускається порівняння значень типу перерахування на рівність чи нерівність, і навіть використання в операторах вибору (у Паскалі – case) як значення, що ідентифікують варіанти.

Використання перерахувань дозволяє зробити вихідні коди програм більш читаними, оскільки дозволяють замінити «магічні числа», що кодують певні значення, на імена, що читаються.

На базі перерахувань у деяких мовах можуть створюватися типи-множини. У цьому разі множина розуміється (і описується) як невпорядкований набір унікальних значень перелічувального типу.

Складні типи даних.

Складні типи даних – це типи, які складаються з елементів, що відносяться до простих типів. До складних типів даних відносяться: структури; класи; об’єкти; масиви; множини; рядки; записи; файли; динамічні змінні; вказівники; лінійні списки (стеки, черги); нелінійні списки (двійкові дерева, несиметричні дерева, тексти, графи); та багато інших.

Складні типи даних (або структури даних) використовуються для групування різних типів даних у один об'єкт, який може представляти складну структуру або об'єкт з багатьма атрибутами. Це дозволяє легше організовувати та обробляти дані, забезпечує модульність і полегшує взаємодію між об'єктами програми.

Складні типи даних є ключовими для роботи зі складними і структурованими даними в програмах. Вони дозволяють впорядковувати дані та робити їх більш легкодоступними та керованими.

Висновки до розділу 2

Історія розвитку систем числення демонструє поступовий перехід від простих способів позначення чисел до більш ефективних методів обчислень. Системи числення поділяються на позиційні та непозиційні. У непозиційних системах, таких як єгипетська або римська, значення символу не залежить від його позиції, що робить арифметичні операції громіздкими. Позиційні системи, де значення цифри визначається її позицією відносно основи, суттєво спрощують обчислення. У комп'ютерних системах використовуються переважно

двійкова, вісімкова та шістнадцяткова системи числення. Двійкова система є основою роботи цифрової електроніки, оскільки два стани 0 і 1 – безпосередньо відповідають фізичним станам транзистора. Вісімкова і шістнадцяткова системи використовуються як компактний запис двійкових чисел – групами по три та чотири біти відповідно.

Цілі числа в пам'яті комп'ютера можуть представлятись у трьох основних форматах. Прямий код є найбільш природним для людського сприйняття, але має суттєві недоліки: два різних представлення нуля і потребу в спеціальних механізмах для виконання арифметичних операцій зі знаковими числами. Обернений код вирішує проблему арифметики, але зберігає два нулі і вимагає циклічного перенесення. Доповняльний код є найефективнішим і найпоширенішим форматом – він усуває обидва недоліки, дозволяє виконувати додавання і віднімання єдиним апаратним пристроєм і має лише один нуль.

Арифметика у двійковій системі виконується за правилами аналогічними десятковій, але з двома цифрами. Множення знакових чисел у доповняльному коді реалізується через алгоритм Бута, що аналізує пари сусідніх бітів множника і замінює послідовне додавання операціями на межах серій одиниць. Ділення є найскладнішою операцією і реалізується через послідовне порівняння і зсув, погано піддається розпаралелюванню і виконується значно повільніше за інші операції.

Обмежена розрядність породжує три основні проблеми: втрату точності через неможливість точного представлення більшості дробів у двійковій системі, підтікання при роботі з числами меншими за мінімально представлюване значення, і накопичення похибок при багаторазових операціях, що підтверджується реальними історичними прикладами.

Числа з плаваючою комою представляються за стандартом IEEE 754 (ISO/IEC 60559) у форматі знак-експонента-мантиса і дозволяють працювати з широким діапазоном значень, що, проте, може бути лише наближеним представленням. Для вирішення задач, де потрібна абсолютна точність використовуються числа з фіксованою комою, наприклад у фінансових розрахунках і вбудованих системах.

Типи даних визначають спосіб інтерпретації бітових послідовностей у пам'яті і повідомляють компілятору, як програміст має намір використовувати дані. Основні машинні типи охоплюють цілі числа різної розрядності зі знаком і без, числа з плаваючою комою, логічні значення, символи і байти.

Питання до самоконтролю

1. Чим позиційна система числення відрізняється від непозиційної? Наведіть приклади.
2. Наведіть загальну формулу позиційного числа та поясніть значення кожної складової.

3. Чому в комп'ютерних системах використовується двійкова система числення, а не десяткова?
4. В чому полягає метод переведення дробової частини десяткового числа у двійкову систему.
5. Яке практичне значення вісімкової системи числення в сучасних операційних системах?
6. Чому два шістнадцяткових розряди відповідають рівно одному байту?
7. Поясніть різницю між прямим, оберненим і доповняльним кодами. Які недоліки прямого коду усуває доповняльний?
8. Чому в прямому та оберненому коді два нулі, а доповняльному – лише один?
9. Чим відрізняється переповнення (overflow) від переносу старшого біта (carry)? Наведіть приклади.
10. Що таке алгоритм Бута і яка його головна ідея? Для яких чисел він застосовується?
11. Поясніть різницю між логічним і арифметичним зсувом праворуч. Для яких чисел використовується кожен?
12. Назвіть основні проблеми обмеженої розрядності.
13. Що таке підтікання (underflow) і чим воно відрізняється від переповнення?
14. Запишіть формулу представлення числа у стандарті IEEE 754 та поясніть призначення кожного поля.
15. Чому числа з плаваючою комою є лише наближенням? Де це є критичним і як з цим борються?
16. Чим числа з фіксованою комою відрізняються від чисел з плаваючою комою?
17. Що таке тип даних і яку роль він відіграє при компіляції програми?
18. Чим тип int відрізняється від unsigned int? Як це впливає на діапазон значень?
19. Для чого використовуються перелічувальні типи даних? Яку перевагу вони дають над використанням числових констант?
20. Чому ділення є найповільнішою арифметичною операцією і як цю проблему вирішують на практиці?

РОЗДІЛ 3. ЛОГІЧНІ ОПЕРАЦІЇ ТА ЇХ ФІЗИЧНА РЕАЛІЗАЦІЯ

3.1. Основні логічні операції: І, АБО, НЕ, XOR.

Логічні операції є основою цифрової електроніки та комп'ютерних обчислень. На апаратному рівні будь-який процесор складається з мільярдів транзисторів, які реалізують саме логічні елементи – НЕ (NOT), І (AND), АБО (OR), виключне АБО (XOR). Ці елементи визначають правила роботи логічних схем, і з них будуються суматори, помножувачі, регістри та інші компоненти процесорів. Вони оперують двійковими значеннями (0 і 1) та використовуються в побудові електронних схем, програмуванні, криптографії та аналізі даних. На програмному рівні логічні операції також мають надзвичайно широке використання. Вони дозволяють ефективно маніпулювати окремими бітами числа – виділяти потрібні розряди, встановлювати або скидати окремі біти, інвертувати їх. Такі операції виконуються за один такт процесора і забезпечують максимальну швидкодію: отримання доповняльного коду, перевірка парності числа, швидке множення на степінь двійки через зсув, упаковка кількох значень в одне ціле число тощо.

Логічне «НЕ» (NOT)

Операція «НЕ» (заперечення, інверсія, позначається як $\neg A$ або $\overline{A}$) перетворює 0 у 1 і навпаки.

A	¬A (NOT A)
0	1
1	0

Логічна операція NOT застосовується для реалізації двійкової логіки на фізичному рівні, де вона втілюється у формі транзисторного інвертора, що змінює полярність сигналу. У мікропроцесорній техніці вона використовується для формування оберненого коду чисел, що є першим кроком до виконання операції віднімання через додавання, а також для вибору активних пристроїв на шині даних (через сигнали «chip select», які часто є інверсними). У програмуванні NOT дозволяє керувати станами булевих прапорців та створювати умови заперечення, що спрощує розгалуження алгоритмів, коли дію потрібно виконати саме за відсутності певної ознаки.

Логічне «І» (AND)

Операція «І» (кон’юнкція, позначається як $A \cdot B$ або $A \wedge B$) є аналогом операції множення, приймає два вхідні сигнали і повертає 1, якщо обидва операнди рівні 1. В іншому випадку результат 0.

A	**B**	**A ∧ B (AND)**
0	0	0
0	1	0
1	0	0
1	1	1

Операція AND використовується у виділенні конкретних фрагментів даних за допомогою бітових масок, що дозволяє процесору «відсікати» зайву інформацію і зчитувати значення окремих бітів у регістрах стану. В архітектурі комп’ютера застосовується для декодування адрес пам’яті та керування дозволами переривань, де вихідний сигнал активується лише при повному збігу всіх необхідних умов. В програмуванні операцію AND використовують для перевірки складних критеріїв у фільтрах даних та для скидання конкретних бітів у нуль без впливу на інші розряди числа.

Логічне «АБО» (OR)

Операція «АБО» (диз’юнкція, позначається як $A + B$ або $A \vee B$) є аналогом операції логічного додавання, і повертає 1, якщо хоча б один із вхідних операндів дорівнює 1.

A	**B**	**A ∨ B (OR)**
0	0	0
0	1	1
1	0	1
1	1	1

Операція OR використовується для об'єднання сигналів від різних джерел в одну лінію та для встановлення («зведення») конкретних бітів у регістрах керування, що дозволяє вмикати певні функції обладнання. На низькому рівні вона допомагає формувати складні керуючі сигнали, де активність хоча б одного

входу має призвести до дії. У програмуванні OR є базовим інструментом для групування опцій (наприклад, прапорців доступу до файлів) та створення гнучких умов, які спрацьовують при виконанні хоча б однієї з кількох альтернатив.

Логічне "Виключне АБО" (XOR)

Операція XOR (eXclusive OR, позначається як $A \oplus B$) є аналогом додавання по модулю 2 в дискретній математиці, і повертає 1, якщо операнди різні, та 0, якщо однакові.

A	B	A ⊕ B (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

Операція XOR є основою арифметичних вузлів процесора, оскільки саме на її властивостях будуються суматори, що виконують додавання двійкових розрядів. Завдяки здатності виявляти розбіжність сигналів, вона використовується для контролю парності при передачі даних, реалізації алгоритмів шифрування та швидкого порівняння двох значень на нерівність. У програмуванні XOR часто застосовується як найбільш енергоефективний спосіб обнулення регістрів (виконання операції над самим собою) та для реалізації простої анімації чи маніпуляцій з графічними буферами, де повторне накладання маски повертає початковий стан пікселів.

Логічні операції реалізуються у вигляді логічних вентилів (логічних елементів) у цифровій електроніці. Ці вентилі є основними конструктивними блоками мікропроцесорів, пам’яті та інших електронних пристроїв.

Окремі логічні вентилі реалізуються за допомогою транзисторів. Наприклад, вентиль AND можна побудувати за допомогою двох транзисторів, з’єднаних послідовно, а вентиль OR – за допомогою двох транзисторів, з’єднаних паралельно.

3.2. Основні закони булевої алгебри

Булева алгебра – це математичний апарат, на основі розглянутих вище логічних операцій, який дозволяє працювати з логічними висловлюваннями та бінарними змінними (операндами), що можуть приймати лише два значення: 0

(false, хибність) або 1 (true, істинність), і, відповідно описувати та перетворювати складні вирази за допомогою чітких правил і законів. Цей розділ було названо на честь британського математика Джорджа Буля, який у 1854 році розробив її основи. Булева алгебра є основою для роботи цифрових схем, логічного програмування, створення алгоритмів і баз даних.

Булева алгебра має набір фундаментальних законів, які визначають правила перетворення логічних виразів:

1. Закон тотожності

$$A \wedge 1 = A \quad A \vee 0 = A$$

Кон'юнкція з 1 і диз'юнкція з 0 не змінюють значення змінної – це нейтральні елементи для відповідних операцій. Аналог множення на 1 і додавання 0 в звичайній арифметиці.

2. Закон поглинання

$$A \wedge 0 = 0 \quad A \vee 1 = 1$$

Кон'юнкція з 0 завжди дає 0, диз'юнкція з 1 завжди дає 1 – незалежно від значення змінної.

3. Закон ідемпотентності

$$A \wedge A = A \quad A \vee A = A$$

Операція змінної з самою собою дає ту саму змінну. На відміну від звичайної арифметики де $A + A = 2A$, у булевій алгебрі повторення не змінює результату.

4. Закон доповнення

$$A \wedge \neg A = 0 \quad A \vee \neg A = 1$$

Кон'юнкція змінної з її запереченням завжди дає 0 – змінна і її інверсія не можуть бути одночасно істинними. Диз'юнкція завжди дає 1 – одне з двох значень завжди істинне. Це закон виключеного третього.

5. Закон подвійного заперечення

$$\neg(\neg A) = A$$

Подвійна інверсія повертає початкове значення. Аналог подвійного мінуса в арифметиці.

6. Закон комутативності

$$A \wedge B = B \wedge A \quad A \vee B = B \vee A$$

Порядок операндів не впливає на результат – так само як у звичайному додаванні і множенні.

7. Закон асоціативності

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$
$$(A \vee B) \vee C = A \vee (B \vee C)$$

Порядок виконання операцій одного типу не впливає на результат – дужки можна розставляти довільно.

8. Закон дистрибутивності

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

Перший закон – аналог розкриття дужок в арифметиці: A·(B+C) = A·B + A·C. Другий закон не має аналога в звичайній арифметиці і є специфічним для булевої алгебри.

9. Закон поглинання

$$A \wedge (A \vee B) = A$$
$$A \vee (A \wedge B) = A$$

Якщо одна зі змінних вже присутня у виразі, додаткова операція з нею не змінює результату. Ці закони дозволяють спрощувати логічні вирази.

10. Закони де Моргана

$$\neg(A \wedge B) = \neg A \vee \neg B$$
$$\neg(A \vee B) = \neg A \wedge \neg B$$

Один з найважливіших законів на практиці. Заперечення кон'юнкції рівнозначне диз'юнкції заперечень, і навпаки. Закони де Моргана дозволяють виражати будь-яку логічну операцію через інші – наприклад, реалізувати OR лише через AND і NOT, що важливо при проектуванні електронних схем де доступні лише певні типи логічних елементів.

Практичне застосування – спрощення умов у програмуванні:

$$\neg(A \wedge B) \rightarrow \neg A \vee \neg B$$

тобто: NOT (умова1 AND умова2) рівнозначно: (NOT умова1) OR (NOT умова2)

11. Закон XOR через AND, OR, NOT

$$A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$$

XOR можна виразити через базові операції – це підтверджує, що AND, OR і NOT утворюють функціонально повну систему, тобто будь-яку логічну функцію можна реалізувати лише через ці три операції.

Всі ці закони використовуються при проектуванні цифрових схем для мінімізації кількості логічних елементів, що безпосередньо впливає на швидкодію і енергоспоживання процесора.

3.3. Логічні елементи та арифметико-логічний пристрій (ALU).

Логічні елементи є фундаментальними компонентами цифрових пристроїв. Вони реалізують операції булевої алгебри та є основою для створення процесорів, контролерів та інших цифрових систем. Використовуючи комбінацію логічних елементів, можна створювати складніші вузли, такі як арифметико-логічний пристрій (ALU), який є основною частиною процесора.

Логічний елемент – це електронний пристрій, що реалізує одну логічну операцію над вхідними сигналами. У сучасних процесорах логічні елементи будуються на основі транзисторів типу КМОН (комплементарна логіка на транзисторах метал-оксид-напівпровідник), де логічний 0 відповідає низькій напрузі (близько 0 В), а логічна 1 – високій (залежно від технології, 1.0–3.3 В).

Основні логічні елементи:

1. Елемент NOT (інвертор). Це найпростіший елемент – один транзистор. При високій напрузі на вході транзистор відкривається і з'єднує вихід із землею (0 В). При низькій напрузі транзистор закритий і вихід підтягується до живлення (1).

2. Елемент AND реалізується через два послідовно з'єднані транзистори – струм проходить лише коли обидва відкриті, тобто обидва входи рівні 1.

3. Елемент OR реалізується через два паралельно з'єднані транзистори – струм проходить якщо хоча б один відкритий.

4. Елемент NAND (І-НІ) та NOR (АБО-НІ). На практиці базовими елементами є не AND і OR, а їх заперечення – NAND і NOR, оскільки вони реалізуються простіше (менше транзисторів) і є функціонально повними системами самі по собі:

NAND: $\neg(A \wedge B)$ – будь-яку функцію можна реалізувати лише через NAND

NOR: $\neg(A \vee B)$ – будь-яку функцію можна реалізувати лише через NOR

Далі розглянемо комбінацію простих логічних елементів у більш складні:

1. Напівсуматор реалізує додавання двох однобітних чисел без урахування вхідного перенесення:

Сума: $A \oplus B$ – один елемент XOR

Перенесення: $A \wedge B$ – один елемент AND

2. Повний суматор враховує вхідне перенесення C_{in} і складається з двох півсуматорів і одного OR:

Сума: $A \oplus B \oplus C_{\text{in}}$

Перенесення: $(A \wedge B) \vee (C_{\text{in}} \wedge (A \oplus B))$

3. Багаторозрядний суматор будується з'єднанням n повних суматорів у ланцюжок – вихідне перенесення кожного розряду подається як вхідне перенесення наступного. Такий суматор називається суматором з послідовним перенесенням і є найпростішою реалізацією, хоча і не найшвидшою – час обчислення зростає лінійно з кількістю розрядів через ланцюжок перенесень.

У сучасних процесорах використовується суматор з прискореним перенесенням де перенесення для кожного розряду обчислюється паралельно за допомогою додаткової логіки, що суттєво збільшує швидкодію.

4. Арифметико-логічний пристрій (АЛП, ALU) – центральний компонент процесора, що виконує всі арифметичні і логічні операції. Його структура складається з кількох основних блоків:

Вхідні регістри – два регістри A і B, що зберігають операнди – числа над якими виконується операція. Вони завантажуються з загальних регістрів процесора перед виконанням операції.

Блок логічних операцій – виконує побітові операції AND, OR, XOR, NOT над вхідними операндами паралельно і одночасно – результат кожної операції завжди обчислений і готовий, залишається лише вибрати потрібний.

Блок арифметичних операцій – містить суматор з прискореним перенесенням як основний елемент. Віднімання реалізується через доповняльний код – АЛП інвертує другий операнд і додає 1 перед подачею на суматор. Множення і ділення реалізуються або окремими апаратними блоками, або через послідовність операцій додавання і зсуву.

Блок зсувів – виконує логічний і арифметичний зсув на задану кількість позицій ліворуч або праворуч. У сучасних процесорах використовується барельний зсувач (Barrel Shifter), що виконує зсув на будь-яку кількість позицій за один такт.

Мультиплексор результату – вибирає результат від одного з блоків залежно від коду операції, що надходить з блоку управління процесора. Саме тут визначається яку операцію виконує АЛП у поточному такті.

Регістр прапорців – зберігає інформацію про результат останньої операції у вигляді окремих бітів – прапорців. Основними прапорцями при роботі з даними є наступні:

Прапорець	**Позначення**	**Значення**
Нуль	Z (Zero)	результат дорівнює нулю
Перенесення	C (Carry)	виникло перенесення з старшого біта
Переповнення	V (Overflow)	результат вийшов за межі діапазону
Знак	N (Negative)	результат від'ємний

Прапорці використовуються командами умовного переходу – саме на їх основі процесор вирішує чи виконувати певну гілку програми (конструкції if, while, for у мовах програмування).

Таким чином арифметико-логічний пристрій є мозком процесора, якому реалізовано функціонування логічних операцій та двійкової арифметики, та який працює на частотах у мільярди тактів на секунду.

Висновки до розділу 3

Логічні операції є фундаментальними для роботи комп'ютерів, програмування та цифрової електроніки. Вони широко застосовуються у мікропроцесорах, операційних системах, комп'ютерних мережах та алгоритмах обробки даних. Логічні вентилі, що реалізують ці операції, складають основу всіх цифрових пристроїв.

Чотири базові логічні операції – NOT, AND, OR і XOR – є основою роботи будь-якого цифрового пристрою. Кожна з них має конкретне математичне визначення і практичне застосування. NOT інвертує сигнал і є першим кроком до отримання доповняльного коду. AND використовується для маскування і виділення бітів. OR дозволяє встановлювати біти і об'єднувати сигнали. XOR є основою суматорів, використовується для виявлення розбіжностей між значеннями, контролю парності і ефективного обнулення регістрів. Принципово важливо, що ці операції виконуються за один такт процесора, що робить їх найшвидшим інструментом маніпуляцій з даними на програмному рівні.

Булева алгебра використовується в багатьох сферах інформаційних технологій: проектування цифрових схем, обробка умов у мовах програмування, маніпуляція бітами в низькорівневому програмуванні, пошукові запити в базах даних, а також в криптографії та шифруванні. Вона дозволяє будувати ефективні логічні схеми та оптимізувати обчислювальні процеси. Розуміння законів і правил булевої алгебри від тотожності і поглинання до законів де Моргана і дистрибутивності – дозволяють спрощувати і перетворювати логічні вирази. Закони де Моргана мають особливе практичне значення: вони доводять, що будь-яку логічну функцію можна виразити через комбінацію AND, OR і NOT, що є теоретичною основою для проектування будь-яких цифрових схем. Мінімізація логічних виразів через ці закони безпосередньо зменшує кількість транзисторів у схемі і впливає на швидкодію та енергоспоживання процесора.

На апаратному рівні логічні операції реалізуються через логічні елементи (вентилі) побудовані на КМОН-транзисторах. Елемент NOT потребує одного транзистора, AND і OR – двох у послідовному і паралельному з'єднанні відповідно. Практично важливим є те, що базовими елементами сучасних мікросхем є не AND і OR, а NAND і NOR – вони реалізуються з меншою кількістю транзисторів і кожен з них є функціонально повною системою, тобто достатнім для реалізації будь-якої логічної функції.

З логічних елементів будуються складніші вузли. Напівсуматор реалізує однобітне додавання через XOR (сума) і AND (перенесення). Повний суматор доповнює його обробкою вхідного перенесення через два напівсуматори і один OR. Багаторозрядний суматор з послідовним перенесенням має лінійну залежність часу від кількості розрядів, тому в сучасних процесорах використовується суматор з прискореним паралельним перенесенням.

Арифметико-логічний пристрій є центральним обчислювальним компонентом процесора і об'єднує блок логічних операцій, блок арифметики з суматором, барельний зсувач і мультиплексор результату. Регістр прапорців фіксує ключові характеристики результату – нуль, перенесення, переповнення і знак – і є основою для реалізації умовних переходів у програмах. Таким чином АЛП є апаратним втіленням всього, що розглядалось у попередніх розділах: логічних операцій, двійкової арифметики і представлення чисел у доповняльному коді.

Питання до самоконтролю

1. Що таке логічна операція і яку роль вона відіграє на апаратному і програмному рівнях комп'ютера?
2. Складіть таблицю істинності для операцій NOT, AND, OR і XOR. Який з результатів є унікальним для XOR порівняно з OR?
3. Поясніть чому XOR двох однакових чисел завжди дає нуль. Як це використовується на практиці в асемблері?
4. Для яких практичних задач використовується операція AND з бітовою маскою? Наведіть приклад.
5. Чим операція OR відрізняється від XOR при встановленні бітів? Наведіть приклад.
6. Яке практичне значення булевої алгебри для сучасної комп'ютерної техніки?
7. Сформулюйте закон подвійного заперечення і поясніть його аналогію з арифметикою.
8. У чому полягає закон ідемпотентності? Чим він відрізняється від аналогічної операції в звичайній арифметиці?
9. Сформулюйте закони де Моргана та наведіть приклад їх застосування для спрощення умовного виразу в програмуванні.
10. Що означає поняття «функціонально повна система» логічних операцій? Чому AND, OR і NOT утворюють таку систему?
11. Чому на практиці базовими елементами мікросхем є NAND і NOR, а не AND і OR? У чому їхня перевага?
12. Скільки транзисторів потрібно для реалізації елементів NOT, AND, OR і XOR?
13. Чим напівсуматор відрізняється від повного суматора?
14. Перелічіть основні блоки АЛП та опишіть їх призначення.
15. Перелічіть основні прапорці регістра прапорців АЛП та поясніть за яких умов встановлюється кожен з них.

РОЗДІЛ 4. ОСНОВИ АРХІТЕКТУРИ СУЧАСНОЇ КОМП’ЮТЕРНОЇ ТЕХНІКИ

4.1. Еволюція архітектури: огляд історичних етапів. Витоки обчислювальних систем: від абака до механічних машин.

Обчислення відігравали ключову роль у розвитку стародавніх цивілізацій, допомагаючи вирішувати практичні завдання в таких сферах, як сільське господарство, будівництво та астрономія. Наприклад, у Месопотамії жерці використовували складну систему числення, засновану на шістдесятковій основі, що дозволяло їм виконувати точні розрахунки. Вони володіли знаннями в галузі геометрії та алгебри, застосовуючи їх для розв’язання різноманітних задач, створювали таблиці множення та розробляли методи обчислення, включаючи розв’язання квадратичних рівнянь.

У Стародавньому Єгипті математика також була тісно пов’язана з повсякденними потребами. Єгиптяни вміли розраховувати площі основних геометричних фігур, таких як прямокутник, трикутник і коло. Відомо, що їхні формули для обчислення площі кола були досить точними, як на той час. Ці математичні знання активно використовувалися у землеробстві для планування іригаційних систем, оцінки врожайності, прогнозування розливів Нілу, а також у будівництві складних архітектурних споруд, включаючи піраміди. Крім того, вони застосовували математичні методи при організації експедицій і військових походів у віддалені регіони.

Одним із перших обчислювальних пристроїв в історії став аба́к (від грецького ἄβαξ – «лічильна дошка»), який використовувався в різних культурах для виконання арифметичних операцій. Він являв собою спеціальну площину, де за певними правилами переміщувалися лічильні елементи. Початково це була дошка, покрита піском або пилом, на якій малювали лінії та розкладали камінці для підрахунків. У Стародавній Греції абак переважно використовувався для грошових розрахунків: у його лівій частині розміщувалися великі номінали, а в правій – дрібні. Рахунок вівся у двійково-п’ятеричній системі числення, що дозволяло швидко виконувати додавання та віднімання шляхом переміщення камінців між розрядами. У Стародавньому Римі цей пристрій зазнав змін: його почали виготовляти з бронзи, слонової кістки або кольорового скла. Римський абак складався з двох рядів прорізів, по яких пересувалися кісточки, що робило його більш зручним у використанні (рис. 4.1). Він дозволяв виконувати складніші обчислення, зокрема працювати з дробовими числами. Римляни називали цей пристрій calculi («камінці»), звідки походить латинське слово calculare («обчислювати»), що згодом дало сучасний термін «калькулятор».

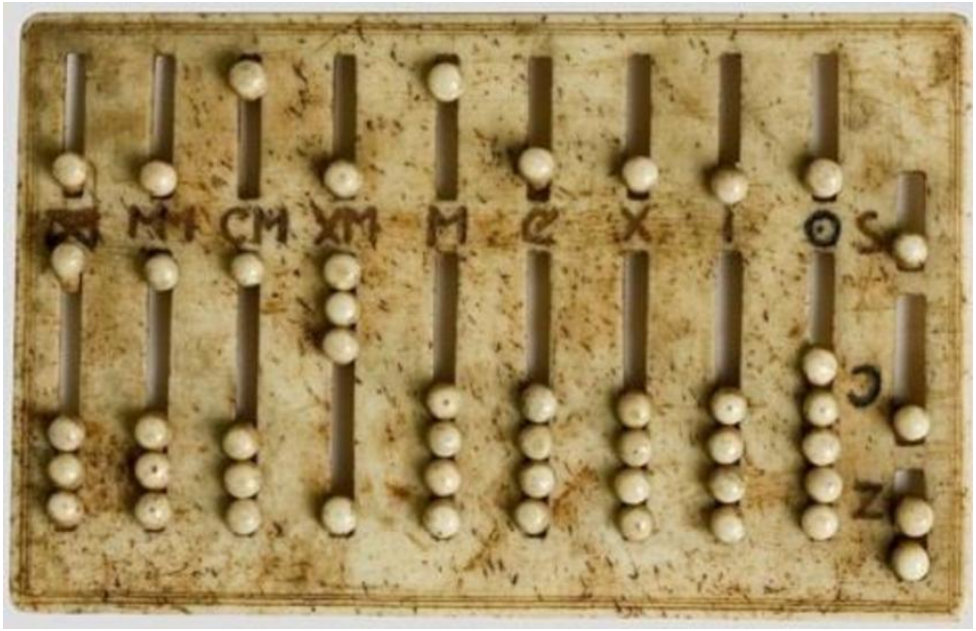

Рисунок 4.1 – Римський абак [10]

Антикітерський механізм, створений у II столітті до н.е., є ще одним видатним прикладом ранніх обчислювальних пристроїв. Цей складний механізм використовувався для прогнозування астрономічних положень і затемнень, демонструючи глибокі знання стародавніх греків у галузі механіки та астрономії. Обчислення здійснювалися за допомогою взаємодії понад 30 бронзових шестерень і декількох циферблатів. Для визначення місячних фаз використовувалася диференціальна передача – механізм, який, як вважалося, був винайдений не раніше XVI століття. Зі занепадом античної епохи технологія створення таких складних пристроїв була втрачена, і знадобилося близько півтори тисячі років, перш ніж людство знову опанувало подібні інженерні рішення. Так, в одному із своїх щоденників Леонардо да Вінчі приводить малюнок тринадцятирозрядного десяткового підсумовуючого пристрою на основі зубчатих коліс.

Таким чином, розвиток механічних обчислювальних пристроїв розпочався з прагнення автоматизувати арифметичні обчислення, що спочатку виконувалися вручну за допомогою простих інструментів, таких як абак. Однак зі збільшенням складності розрахунків, особливо у сфері астрономії, торгівлі та фінансів, виникла необхідність у створенні більш досконалих механізмів.

Одним із перших значних кроків у цій галузі стала "арифметична машина" Вільгельма Шиккарда, створена в 1623 році. Це був механічний калькулятор, здатний виконувати додавання та віднімання, а за допомогою спеціальних механізмів– множення та ділення. Пристрій мав зубчасті колеса та нагадував годинниковий механізм. На жаль, оригінальний зразок був втрачений, а винахід Шиккарда залишався маловідомим аж до XX століття.

Наступним проривом став "Паскалін", створений у 1642 році Блезом Паскалем. Це був перший механічний калькулятор, який міг виконувати

автоматичне додавання та віднімання шляхом обертання зубчастих коліс. Паскалін широко використовувався французькими фінансистами для обліку та фінансових операцій.

Розвиток механічних обчислювальних пристроїв продовжив Готфрід Вільгельм Лейбніц, який у 1673 році створив "Степеневий калькулятор", або арифмометр (Stepped Reckoner). На відміну від попередників, ця машина могла виконувати всі чотири арифметичні операції (додавання, віднімання, множення, ділення) та використовувала "ступінчасте колесо Лейбніца", яке згодом стало основою для багатьох механічних калькуляторів XIX століття.

Подальший прогрес у цій галузі відбувся завдяки Чарльзу Беббіджу (Charles Babbage), який у 1822 році розпочав розробку "Диференційної машини" (Difference Engine) – пристрою, здатного автоматично обчислювати таблиці чисел методом кінцевих різниць. Надалі він розробив ще більш амбітний проєкт – "Аналітичну машину" (Analytical Engine) (рис.4.2), яка містила всі основні компоненти сучасного комп’ютера: арифметичний блок (прообраз ALU), пам’ять для зберігання даних і перфокартний механізм для введення програм. Цей проєкт значно випереджав свій час, і через технічні обмеження XIX століття машина так і не була повністю зібрана.

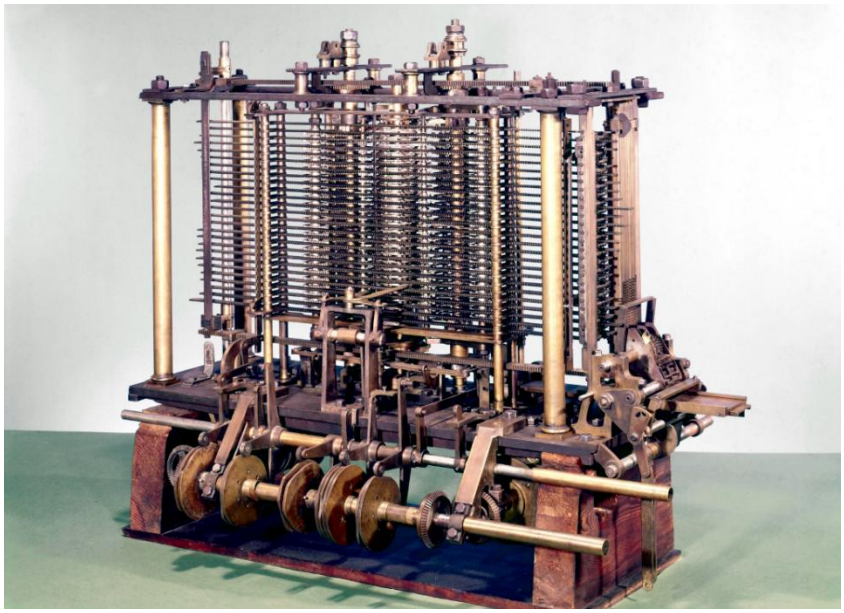

Рисунок 4.2 – Аналітична машина Бебіджа [11]

Ада Лавлейс, співпрацюючи з Беббіджем, написала три обчислювальні програми для аналітичної машини Чарльза Беббіджа. Перша програма була написана для здійснення рішень двох рівнянь алгебри з двома невідомими, тут вона ввела вперше класифікацію осередків пам'яті, що застосовується в програмуванні й в наш час, як і термін "робочий осередок". У другій програмі, призначеної для обчислення значень тригонометричної функції з неодноразовим повторенням заданої послідовності, озвучила поняття "цикл". Найскладнішою

виявилася третя програма – програма для обчислення відомих у математиці чисел Бернуллі. Вона написала покрокову інструкцію для обчислення чисел Бернуллі на аналітичній машині. Результат її роботи залишився в історії як перша надрукована робота з програмування.

У XIX столітті механічні калькулятори вдосконалювалися та набули широкого комерційного застосування. Серед найпопулярніших пристроїв були арифмометри, засновані на принципах Лейбніца, які масово використовувалися до середини XX століття.

Початок XX століття ознаменувався появою електромеханічних обчислювальних пристроїв, які поєднували механічні компоненти з електричними схемами. Одним із найважливіших винаходів став Mark I (Говард Айкен, 1944) – перший програмований електромеханічний комп’ютер, який працював на основі релейної логіки.

Подальший розвиток обчислювальних систем призвів до створення першого покоління електронних комп’ютерів у 1940-х роках, таких як ENIAC (1946), який став першою повністю електронною цифровою обчислювальною машиною. Відхід від механічних компонентів та використання електронних ламп започаткували нову епоху в розвитку комп’ютерної техніки.

Таким чином, шлях від перших механічних калькуляторів до електронних комп’ютерів тривав понад три століття та включав низку важливих відкриттів і винаходів, які заклали основу сучасних обчислювальних систем.

Покоління обчислювальних машин. Еволюція технічних засобів.

Поділ обчислювальних машин на покоління є зручним способом класифікації їх розвитку та еволюції. Основним критерієм переходу від одного покоління до іншого є впровадження принципово нових технологій, які змінювали продуктивність, надійність, розміри та сферу застосування обчислювальних систем. Основні критерії такого поділу можна розглянути з точки зору апаратного забезпечення, програмного забезпечення, архітектури та сфери використання. Традиційно, одним з основних факторів, що визначав зміну поколінь комп’ютерів, була технологія елементної бази, тобто тип компонентів, на яких будувалися обчислювальні та запам’ятовуючі пристрої, з кожною новою технологією відбувався стрибок у швидкодії, зменшення розмірів, енергоефективності та вартості обчислювальних машин. Іншим важливим критерієм для визначення поколінь стала еволюція архітектури комп’ютерів та те, як змінювалися засоби програмування та взаємодії з комп’ютерами. Також якщо на ранніх етапах комп’ютери були гігантськими машинами, доступними лише військовим, науковим установам і великим корпораціям, то з розширенням можливостей комп’ютерів, зменшення вартості та зростання попиту на автоматизацію різних сфер життя вони стали доступними для бізнесу, а потім – для персонального використання.

Перше покоління (1940–1950-ті роки): Ера електронних ламп.

Перше покоління комп’ютерів стало проривним у розвитку обчислювальної техніки, оскільки вперше були створені повністю електронні обчислювальні машини, які замінили механічні та електромеханічні пристрої. Головною відмінністю цього покоління стала елементна база – обчислення виконувалися за допомогою електронних ламп (вакуумних діодів і тріодів), що значно підвищило швидкість обробки інформації. Однак ці машини були величезними, дорогими та вимагали великих витрат електроенергії. Вони застосовувалися переважно у військовій сфері, наукових дослідженнях і державному управлінні. Програмування таких машин було складним, оскільки воно здійснювалося за допомогою машинних кодів або безпосередньої перекомутації електричних схем. Перехід до електронних комп’ютерів став можливим завдяки винаходу електронних ламп (1904, Джон Флемінг), які дозволили створювати швидкі й надійні логічні схеми.

Технічні особливості комп’ютерів першого покоління

– Елементна база: електронні лампи (від кількох тисяч до десятків тисяч).

– Оперативна пам’ять: ртутні лінії затримки, електростатичні трубки Вільямса, магнітні барабани.

– Запам’ятовувальні пристрої: перфокарти, перфострічки.

– Програмування: машинний код, ручне комутування з’єднань.

– Операційні системи: відсутні (кожна програма писалася вручну).

– Швидкодія: кілька тисяч операцій за секунду.

– Розміри: займали цілі приміщення.

– Споживання електроенергії: десятки кіловат.

На ранньому етапі обчислювальні машини мали різні архітектурні рішення, але ключовою подією стала архітектура фон Неймана (1945), яка визначила стандартну модель комп’ютера, що була застосована в більшості машин першого покоління:

– Дані та програми зберігаються в одній пам’яті.

– Виконання інструкцій здійснюється послідовно, одна за одною.

– Використовується арифметично-логічний пристрій (ALU) для обчислень.

– Єдина шина для обміну даними між компонентами.

До основних представників комп’ютерів першого покоління можна віднести наступні:

1. ENIAC (1946, США) (рис. 4.3). Розробники: Джон Моклі, Преспер Еккерт (Філадельфія). Кількість електронних ламп: 17 468. Оперативна пам’ять: 20 акумуляторів цифр (резисторно-ємнісна пам’ять). Програмування: перемикання тумблерів і перекомутовування кабелів. Продуктивність: 5000 додавань або 357 множень за секунду. Призначення: розрахунок артилерійських таблиць, моделювання ядерних вибухів.

2. EDSAC (1949, Велика Британія) Перша машина, що працювала за архітектурою фон Неймана. Розробник: Моріс Вілкс (Кембридж). Оперативна пам’ять: ртутні лінії затримки. Призначення: наукові обчислення.

3. МЭСМ (1950, УРСР) Перша електронна обчислювальна машина в Україні. Розробник: Сергій Лебедєв (Київ). Використання: моделювання фізичних процесів, наукові розрахунки.

4. UNIVAC I (1951, США) Перший комерційний комп’ютер. Розробники: Джон Моклі, Преспер Еккерт. Замовник: Бюро перепису США. Використання: економічні та статистичні розрахунки.

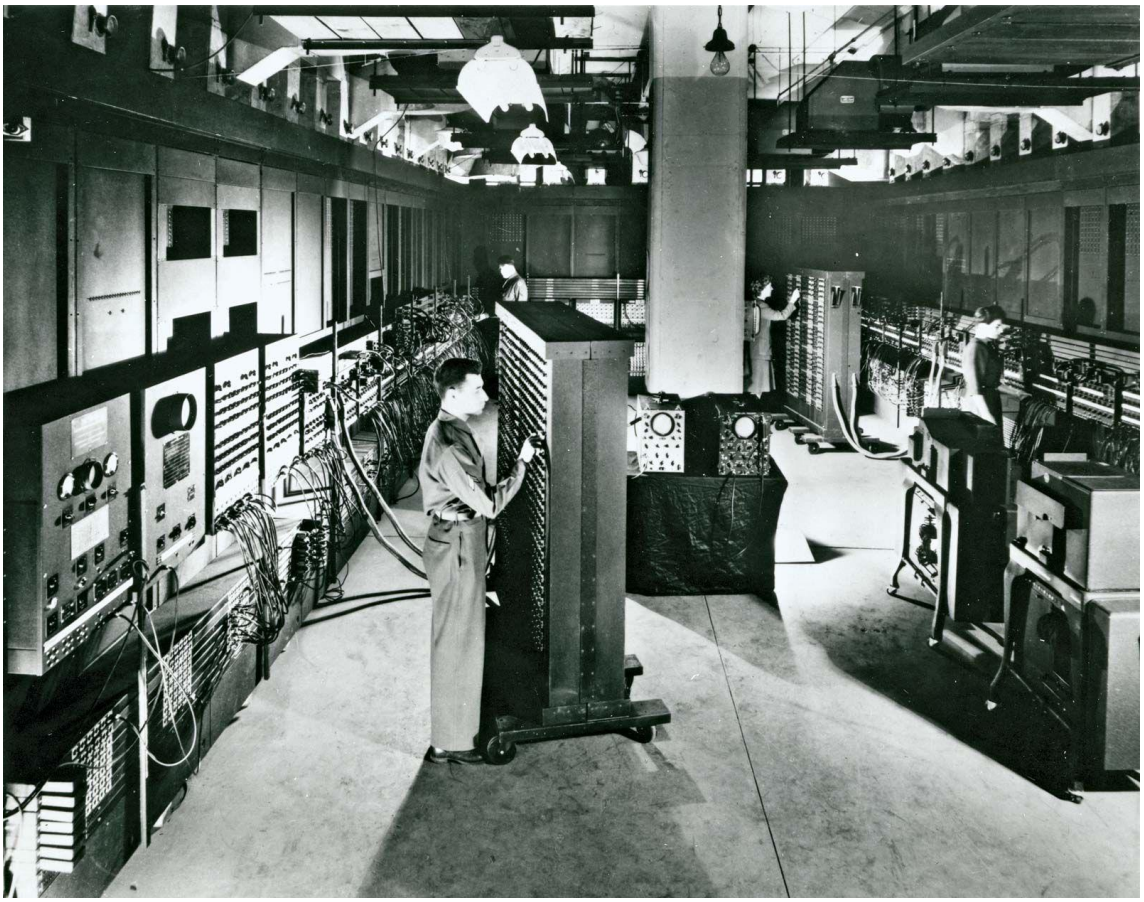

Рисунок 4.3 – ЕОМ першого покоління ENIAC [12]

До переваг першого покоління над попередніми обчислювальними пристроями можна віднести наступні:

– Електронні лампи забезпечили суттєве збільшення швидкодії.

– Можливість автоматичного виконання складних обчислень.

– Перші спроби створення універсальних програмованих машин.

Відповідно до недоліків таких обчислювальних машин можна віднести наступні:

– Громіздкість і високе енергоспоживання.

– Низька надійність – електронні лампи часто виходили з ладу.

– Складність програмування – відсутність мов програмування високого рівня, а саме програмування виконувалось вручну перемиканням тумблерів і перекомутацією кабелів.

– Відсутність операційних систем.

На початку 1950-х стало зрозуміло, що електронні лампи мають серйозні обмеження. У 1947 році Джон Бардин, Вільям Шоклі та Волтер Браттейн винайшли транзистор, який став революційним рішенням. Завдяки меншому розміру, енергоспоживанню та надійності, транзистори почали поступово замінювати електронні лампи, що стало основою для другого покоління комп’ютерів.

Друге покоління (1950–1960-ті роки): Ера транзисторів.

Перше покоління мало серйозні недоліки, які стримували розвиток обчислювальної техніки: ненадійні у використання електронні лампи, величезні розміри та надмірне споживання електроенергії.

Вирішенням цих проблем став винахід транзистора (1947, Джон Бардин, Вільям Шоклі, Волтер Браттейн, Bell Labs). З появою масового виробництва транзисторів у 1950-х комп’ютери стали значно меншими, швидшими та надійнішими.

Ще однією ключовою зміною стало впровадження мов програмування високого рівня, зокрема Fortran (1957) і COBOL (1959), що зробило комп’ютери більш доступними для програмістів. Паралельно з'явилися операційні системи, які дозволили розширити можливості взаємодії користувачів з комп’ютерами. Якщо перше покоління було орієнтоване переважно на наукові та військові розрахунки, то друге покоління активно проникло в бізнес та адміністративні процеси, що зробило комп’ютери більш універсальними.

Друге покоління обчислювальних машин стало якісним стрибком у розвитку комп’ютерної техніки завдяки заміні електронних ламп транзисторами. Це дозволило суттєво зменшити розміри, знизити енергоспоживання та підвищити надійність машин.

Технічні особливості комп’ютерів другого покоління:

– Елементна база: транзистори замість електронних ламп.

– Оперативна пам’ять: феритові осередки (феритно-магнітна пам’ять).

– Запам’ятовувальні пристрої: магнітні барабани, магнітні стрічки, перші жорсткі диски.

– Програмування:

- Мови високого рівня: Fortran (1957), COBOL (1959), ALGOL (1958).

- Використання асемблерів (мови, що дозволяли працювати з командами процесора).

– Операційні системи: з’явилися перші ОС (IBM OS/360, BESYS).

– Швидкодія: десятки тисяч – сотні тисяч операцій за секунду.

– Розміри: у кілька разів менші за комп’ютери першого покоління.

– Споживання електроенергії: у десятки разів менше, ніж у лампових комп’ютерів.

До основних представників комп’ютерів другого покоління можна віднести наступні:

1 IBM 1401 (1959, США) (рис. 4.4)– перший масовий комерційний комп’ютер для бізнесу та банківської сфери. Розробник: IBM. Технічні характеристики: використання транзисторів замість ламп; ОЗП: 4–16 КБ (феритні осередки); магнітні стрічки для зберігання даних; швидкодія: 16 000 операцій за секунду.

2 UNIVAC 1108 (1964, США) – призначався для обробки великих обсягів даних у промисловості та науці. Розробник: Sperry Rand Corporation. Технічні характеристики: використання транзисторної логіки, пам’ять: феритові осередки та магнітні стрічки. Особливість: перший уніфікований набір команд (сумісність між різними моделями).

3 БЭСМ-6 (1968, СРСР) – призначення: наукові обчислення, військові дослідження, аерокосмічна галузь. Розробник: Інститут точної механіки і обчислювальної техніки (УРСР), Сергій Лебедєв. Технічні характеристики: використання транзисторів (понад 60 000 штук); ОЗП: 64 КБ (на феритових осередках); швидкодія: до 1 млн операцій за секунду.

Рисунок 4.4 – IBM 1401 [13]

До переваг другого покоління над першим можна віднести наступні:

– Менші розміри – комп’ютери стали в десятки разів компактнішими.

– Зниження енергоспоживання – у десятки разів менше електроенергії.

– Значно вища продуктивність – швидкодія зросла у 10–100 разів.

– Підвищена надійність – транзистори працювали стабільно, на відміну від ламп.

– Поява мов програмування високого рівня – Fortran, COBOL.

– Перші операційні системи – автоматизація процесів керування комп’ютером.

Відповідно, до недоліків таких обчислювальних машин можна віднести:

– Все ще великі розміри та висока вартість – хоч і менші за попередників, комп’ютери все ще коштували сотні тисяч доларів.

– Обмежена пам’ять – оперативна пам’ять складалася з феритових осередків і була дуже дорогою.

– Відсутність інтегральних схем – елементи схеми з’єднувалися вручну, що обмежувало розвиток комп’ютерів.

Попри переваги другого покоління, існували певні технологічні обмеження: комп’ютери все ще залишалися дорогими і не могли масово використовуватися в малому бізнесі та побуті; проводове з’єднання тисяч транзисторів створювало складнощі у виробництві, підвищувало вартість і зменшувало надійність, а також була очевидною необхідність подальшого підвищення продуктивності та зменшення розмірів машин.

Рішенням стало винайдення Джеком Кілбі (Texas Instruments) та Робертом Нойсом (Fairchild Semiconductor) інтегральних схем (IC) у 1958–1959 роках, що дозволило розміщувати сотні транзисторів на одному чіпі. Це стало основою для третього покоління комп’ютерів, яке зробило машини ще компактнішими, продуктивнішими та доступнішими.

Третє покоління (1960–1970-ті роки): Ера інтегральних схем.

Попри значні покращення другого покоління (перехід на транзистори), все ще залишалися проблеми, які вимагали вирішення: Велика складність збірки комп’ютерів, обмежені можливості оперативної пам’яті, необхідність у стандартизації.

Рішенням стало винайдення інтегральних схем: 1958 – Джек Кілбі (Texas Instruments) створив перший прототип IC. 1959 – Роберт Нойс (Fairchild Semiconductor) розробив інтегральні схеми на кремнієвій основі. 1964 – IBM випускає серію System/360, що стала стандартом у світі мейнфреймів.

Інтегральні схеми дозволили зменшити розміри комп’ютерів у десятки разів, а також зробили їх дешевшими та надійнішими, що сприяло їхньому масовому поширенню.

Ще одним важливим досягненням стало уніфікування архітектури – виробники почали випускати серії комп’ютерів з однаковими інструкціями та можливістю розширення. З’явилися багатозадачні операційні системи, а мови програмування стали більш доступними та потужними.

Так, вперше комп’ютери почали використовуватися не лише в урядових і наукових установах, а й у промисловості, медицині та бізнесі, відкриваючи шлях до майбутньої персоналізації обчислювальної техніки.

Третє покоління обчислювальних машин стало справжнім проривом завдяки впровадженню інтегральних схем (IC, Integrated Circuits), які дозволили розміщувати сотні транзисторів на одному кристалі кремнію. Це суттєво зменшило розміри комп’ютерів, знизило їхню вартість і підвищило продуктивність.

Технічні особливості комп’ютерів третього покоління:

– Елементна база: інтегральні схеми (IC) замість транзисторів.

– Оперативна пам’ять: феритові осередки, перші напівпровідникові чипи.

– Запам’ятовувальні пристрої: жорсткі диски (HDD), магнітні стрічки, магнітні картки.

– Програмування: мови високого рівня: Fortran IV, COBOL, ALGOL-68, PL/I.

– Розширення операційних систем: OS/360, UNIX (1969).

– Швидкодія: до мільйонів операцій за секунду.

– Розміри: у десятки разів менші за друге покоління.

– Споживання електроенергії: у сотні разів менше порівняно з ламповими комп’ютерами.

– Багатозадачність: можливість виконувати декілька програм одночасно.

До основних представників комп’ютерів третього покоління можна віднести наступні:

1 IBM System/360 (1964, США) (рис. 4.5) – вперше використовувався в бізнесі, науці, банківській сфері. Розробник: IBM. Технічні характеристики: використання інтегральні схеми. ОЗП: до 8 МБ (на феритових осередках). Підтримка багатозадачності. Стандартна архітектура – моделі могли масштабуватися за потребами користувача.

2 PDP-8 (1965, США) – перший мінікомп’ютер, доступний малим компаніям. Призначення: лабораторні дослідження, промисловість, управління процесами. Розробник: Digital Equipment Corporation (DEC). Технічні характеристики: пам’ять: до 32 КБ. Використання мови асемблера та Fortran.

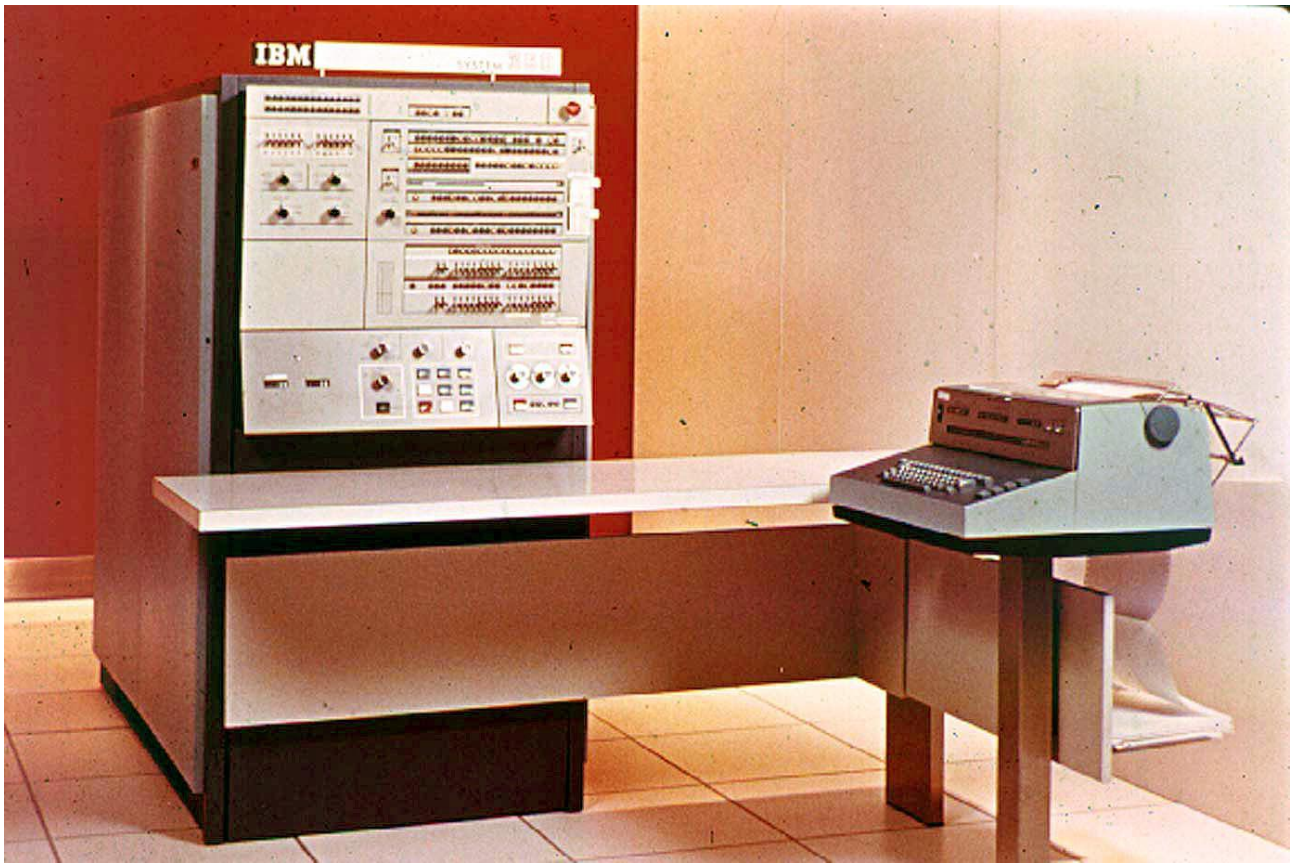

Рисунок 4.5 – IBM System/360 [14]

До переваг третього покоління над другим можна віднести наступні:

– Комп’ютери стали доступнішими – з’явилися мінікомп’ютери, що могли дозволити собі малі компанії.

– Розвиток ОС – операційні системи стали складнішими, багатозадачними.

– Швидший розвиток програмування – з’явилися більш універсальні мови.

– Підвищена продуктивність – до мільйона операцій за секунду.

– Менші витрати на виробництво – інтегральні схеми суттєво здешевили комп’ютери.

Відповідно, до недоліків таких обчислювальних машин можна віднести:

– Все ще висока вартість для персонального використання.

– Потреба в подальшій мініатюризації – навіть мінікомп’ютери займали багато місця.

– Обмеження в швидкодії – незважаючи на покращення, обчислення вимагали ще більшої продуктивності.

Попри всі досягнення третього покоління, інженери розуміли, що інтегральні схеми можна ще більше покращити: У 1971 році Intel представила перший мікропроцесор (Intel 4004), який містив 2300 транзисторів на одному чипі. З’явилася динамічна оперативна пам’ять (DRAM), що замінила феритові

осередки. Персональні комп’ютери все ще не існували, оскільки великі машини залишалися дорогими.

Рішенням цих проблем став перехід до мікропроцесорної архітектури, що стало основою четвертого покоління обчислювальних машин.

Четверте покоління (1970-ті – сучасність): Ера мікропроцесорів.

Попри значні покращення третього покоління, все ще залишалися серйозні проблеми, які вимагали вирішення: висока вартість обчислювальних машин, обмежені можливості інтегральних схем – для підвищення продуктивності потрібно було об’єднувати тисячі мікросхем, що ускладнювало виробництво та ремонт, та складність програмування.

Головним технологічним проривом стало створення мікропроцесора – пристрою, що поєднував тисячі, а згодом мільйони транзисторів на одному кристалі, що дозволило значно зменшити розміри, вартість і енергоспоживання комп’ютерів. Завдяки цьому з'явилися персональні комп’ютери (PC), які стали доступними не лише для великих компаній, а й для окремих користувачів. Програмне забезпечення також зробило великий крок уперед: графічні інтерфейси, багатозадачні операційні системи, мови програмування високого рівня, які значно спростили роботу з комп’ютерами. Це покоління стало переломним: комп’ютери перестали бути лише інструментами в лабораторіях і банках – і потрапили в кожну оселю, кишеню й автомобіль. Водночас, розвиток процесорної архітектури й інтегральних схем привів до експоненційного зростання продуктивності, що вплинуло на всі сфери людської діяльності.

Основними особливостями розвитку комп’ютерної техніки, зокрема процесорів на цьому етапі були:

– Елементна база: мікропроцесори (CPU) на основі MOS-транзисторів.

– Оперативна пам’ять: динамічна (DRAM), статична (SRAM).

– Запам’ятовувальні пристрої: жорсткі диски (HDD), оптичні диски (CD/DVD), флеш-пам’ять.

– Програмування: мови високого рівня: C, Pascal, BASIC, C++, Java.

– Операційні системи: MS-DOS, UNIX, Windows (1985).

– Графічний інтерфейс: GUI (графічний користувацький інтерфейс).

– Мережеві технології: розвиток Інтернету, локальних мереж (LAN).

– Швидкодія: від мільйонів до мільярдів операцій за секунду.

– Компактність: комп’ютери стали доступними для особистого використання.

– Багатозадачність: можливість одночасного виконання кількох програм.

В цей час мікропроцесори стали центральними елементами обчислювальних систем. Усі основні функції процесора реалізовані на одному чипі. Почалось впровадження архітектури RISC (Reduced Instruction Set

Computer): спрощений набір інструкцій, одноклокові операції, великий набір регістрів; Використання динамічної оперативної пам’яті (DRAM) та кеш-пам’яті; активний розвиток та впровадження графічних прискорювачів (GPU), згодом – паралельних обчислень на GPU, так на початку 2000-х років з’явилися багатоядерні процесори (початок 2000-х). Зросла інтеграція мережевих інтерфейсів, високошвидкісних шин (PCIe), USB, NVMe та розпочалось масове впровадження мобільних платформ: смартфони, планшети, вбудовані системи.

До основних представників комп’ютерів четвертого покоління можна віднести наступні:

1 Intel 4004 (1971, США) – призначався для використання калькуляторів, вбудованих систем. Розробник: Intel. Технічні характеристики: 2300 транзисторів. Швидкодія: 60 000 операцій за секунду. Розрядність: 4 біти.

2. Intel 8086 / IBM PC (1978, США) (рис. 4.6) – цей комп’ютер заклав початок епохи персональних комп’ютерів, їх широке застосування в офісах, при розробці та навчанні. Технічні характеристики: процесор: Intel 8086 (16-бітний CISC), операційна система: MS-DOS. До особливостей цього комп’ютера можна віднести те, що він став родоначальником однойменної архітектури x86, що стала стандартом на декілька десятиліть.

Рисунок 4.6 – IBM PC з процесором Intel 8086 [15]

3. Apple Macintosh (1984, США) – перший масовий комп’ютер із GUI. Розробник: Apple Inc. Технічні характеристики: Мікропроцесор: Motorola 68000 (8 МГц), ОЗП: 128 КБ, ОС: Mac OS (графічний інтерфейс).

4. MIPS R2000 (1985, США, Stanford University). – один з перших комп’ютерів на базі архітерктури RISC, що використовувався для робочих станцій, вбудованих систем та мережевого обладнання. Технічні характеристики: процесор: MIPS-I (RISC); кількість транзисторів: ~110 000.

Особливості: чітко визначена структура команд, одна інструкція – один такт виконання, реалізація суперскалярних і конвеєрних обчислень.

5. ARM7TDMI / ARM Cortex (1993–2000+, Велика Британія) – перші енергоефективні пристрої на базі архітектури RISC, що використовувались для мобільних пристроїв, вбудованих та IoT-систем. Особливостями цих процесорів стали: мала площа кристала, низьке енергоспоживання та використання ліцензованої моделі розповсюдження, що дозволило охопити широкий широкий ринок мобільних та IoT-пристроїв та систем.

До переваг четвертого покоління можна віднести наступні:

– Мікропроцесори значно зменшили розміри комп’ютерів.

– Уніфікація обчислювальної техніки – один і той же процесор може бути і в ПК, сервері й холодильнику.

– Масовість і доступність – персональні комп’ютери стали реальністю для мільйонів користувачів.

– Розвиток операційних систем, багатозадачності, графічного інтерфейсу.

– З’явилися комп’ютерні мережі, Інтернет, бази даних.

– Висока продуктивність – завдяки збільшенню тактової частоти, багатоядерності, паралелізму.

– Розвиток архітектури RISC – спрощення конвеєрів, краща енергетична ефективність.

– Поява спеціалізованих прискорювачів (GPU, DSP), що спричинило революція у графіці, мультимедіа та науці.

Відповідно поява певних недоліків як продовження переваг:

– Велика кількість інструкцій і нестандартизованість (особливо в CISC): складність оптимізації коду.

– Залежність від пропускної здатності пам’яті – так званий bottleneck вузьке місце пропускної здатності.

– Теплові обмеження – зростання кількості транзисторів вимагало активного охолодження.

– Ускладнення програмування для вбудованих платформ (ARM, MIPS) через обмеження ресурсів.

Отже, четверте покоління комп’ютерів, епоха мікропроцесорів, сформувало сучасний вигляд обчислювальних систем, поєднала в собі еволюцію від x86 до багатоядерних CPU, революцію RISC, сприяло масовому поширенню персональних комп’ютерів, мобільних пристроїв та мереж, стало підґрунтям для сучасних гетерогенних, розподілених і високопродуктивних обчислень. Саме це покоління створило фундамент інформаційного суспільства, у якому ми живемо сьогодні.

Також розглянемо передумови для подальшого розвитку обчислювальної техніки:

- Обмеження закону Мура: після 2005–2010 рр. темп подвоєння транзисторів уповільнився.
- Проблема теплової стелі: зростання частоти роботи процесорів та шин має певні обмеження, пов’язані з тепловідведенням.
- Набуває актуальності потреба в нових типах архітектури, таких, що забезпечать розвиток паралелізму обчислень, енергоефективність, обчислення в пам’яті тощо.
- Поява високоспеціалізованих задач: нейронні мережі, квантова криптографія, обробка великих даних.
- Зростає інтерес до гетерогенних архітектур (поєднання CPU+GPU, CPU+FPGA).

Закон Мура – це емпіричне спостереження, зроблене Гордоном Муром (співзасновником Intel) у 1965 році, згідно з яким кількість транзисторів на мікросхемі подвоюється приблизно кожні 18–24 місяці (рис. 4.7), що веде до зростання продуктивності процесорів та зменшення їхньої вартості. Подивимось на історію:

– 1960-ті – 1980-ті – закон діє безвідмовно: транзистори стають дедалі меншими, швидшими та дешевшими.

– 1990-ті – 2000-ті – досягнуто нанометрових масштабів (Intel Pentium 4 – 42 млн транзисторів у 2004 році).

– 2010-ті – 2020-ті – закон Мура починає уповільнюватися через фізичні обмеження транзисторів (<10 нм).

– Сучасність – традиційне подвоєння кількості транзисторів стикається з викликами, що потребують нових рішень.

Чому закон Мура наближається до межі?

– Фізичні обмеження – транзистори наблизилися до розміру атомів (менше 3 нм).

– Зростання тепловиділення – подальше підвищення тактової частоти обмежене розсіюванням тепла.

– Зростання вартості виробництва – створення чипів нових поколінь стає дорожчим.

Отже, закон Мура протягом десятиліть залишався основним символом розвитку процесорів, але зараз він поступово втрачає актуальність. Замість класичного подвоєння транзисторів, індустрія шукає нові підходи до підвищення продуктивності, такі як 3D-структури чипів, RISC-архітектури, квантові та нейроморфні обчислення.

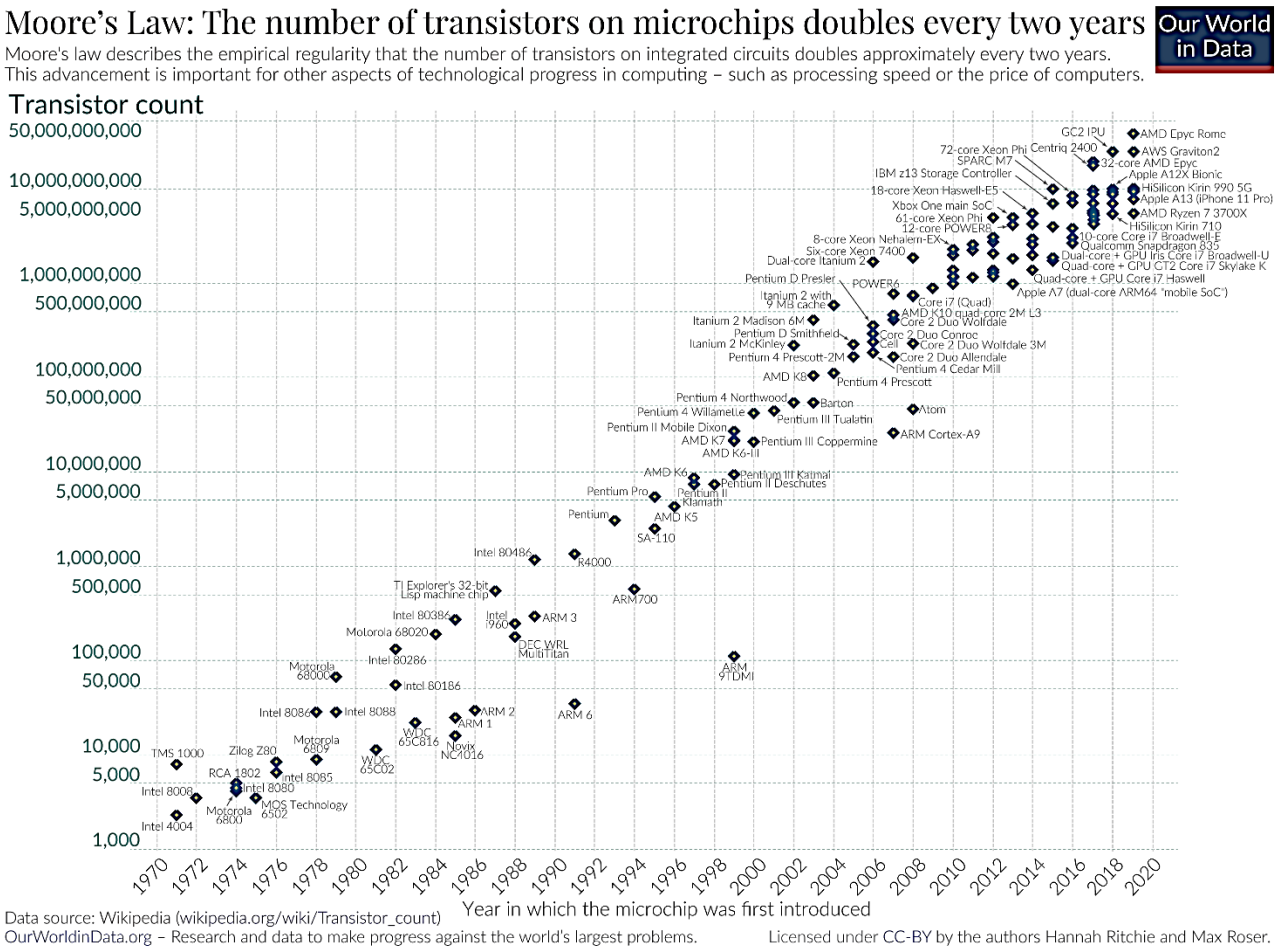

Рисунок 4.7 – Кількість транзисторів у мікропроцесорі в залежності від дати їх появи (наочна демонстрація закону Мура)

Майбутні покоління: квантові, нейроморфні, оптичні обчислення

Класичні процесори вже не справляються з деякими задачами – зокрема, квантовими симуляціями, криптографією та системами штучного інтелекту, а суперкомп’ютери споживають величезну кількість електроенергії, і до того ж виникає потреба в нових підходах до обчислень – класична фон-нейманівська архітектура має вузькі місця, які стримують продуктивність обчислювальних систем.

Наступні покоління комп’ютерів будуть відзначатись переходом від традиційних транзисторних обчислень до принципово нових технологій. Основними нововведеннями цього покоління можуть стати:

- Квантові комп’ютери – обчислення на основі квантових бітів (Q-бітів, або кубітів), що може викликати експоненційне зростання продуктивності в певних задачах, таких як криптографія та складні математичні системи, машинне навчання та штучний інтелект, складні фізичні та хімічні процеси.
- Нейроморфні процесори – моделювання роботи людського мозку для створення енергоефективних систем штучного інтелекту.

- Оптичні та фотонні комп’ютери – використання світла замість електронів для обчислень, що значно підвищує швидкодію при зменшені тепловіддачі.
- Молекулярні та ДНК-комп’ютери – використання біологічних молекул для обчислень.
- Гібридні архітектури – інтеграція квантових, фотонних, нейроморфних та біокомп’ютерів у єдину систему.

Окрема варто виділити системи штучного інтелекту (AI) та машинне навчання – активне використання глибоких нейромереж для аналізу великих даних та автоматизації процесів.

На відміну від попередніх поколінь, такі комп’ютери не просто підвищать продуктивність, але й змінять фундаментальні принципи обчислень, відкриваючи шлях до нестандартних архітектур та нових методів обробки інформації.

Основними подіями розвитку наступного покоління можна назвати наступні:

1994 – Пітер Шор розробляє квантовий алгоритм факторизації, що може загрожувати традиційній криптографії.

2002 – вперше розроблено ДНК-комп’ютер (Леонард Адлеман, США).

2014 – IBM представляє нейроморфний чип TrueNorth.

2018 – IBM створює перший 50-кубітний квантовий процесор.

2019 – Google досягає квантової переваги (комп’ютер Sycamore вирішує задачу, яку класичні суперкомп’ютери не можуть виконати).

Таким чином, наступні покоління мають подолати межі традиційних транзисторних комп’ютерів. Серед існуючих технологій та особливостей таких комп’ютерів можна виділити наступні:

Елементна база:

- Квантові процесори (кубіти, надпровідні ланцюги, іонні пастки).
- Нейроморфні чипи (архітектури, що імітують нейрони).
- Фотонні обчислення (використання світлових сигналів замість електронних).

Оперативна пам’ять:

- Традиційна DRAM та оптична пам’ять.
- Квантова пам’ять (для спеціальних задач).

Запам’ятовувальні пристрої:

- 3D XPoint, MRAM, RRAM (нові типи пам’яті з надзвичайною швидкістю запису/зчитування).
- Хмарні технології – дані не зберігаються локально, а розподіляються в дата-центрах.

Програмування:

- Квантові мови: Q, Qiskit, Cirq.
- Штучний інтелект та машинне навчання: TensorFlow, PyTorch.
- Гібридні обчислення: поєднання традиційних та квантових процесорів.

Інші технології:

- Розвиток штучного інтелекту (AI) – розпізнавання мови, зображень, автономні системи.
- Інтернет речей (IoT) – мільярди пристроїв, з’єднаних у глобальну мережу.
- Розвиток блокчейну – децентралізовані технології збереження та передачі даних.

Рисунок 4.8 – Квантовий комп’ютер (процесор і система охолодження) [16]

До основних представників комп’ютерів шостого покоління можна віднести наступні:

1. Квантові комп’ютери (рис. 4.8):

1.1. IBM Quantum Hummingbird (2020, США). Призначення: моделювання молекулярних процесів, криптографія, оптимізація. Розробник: IBM. Кількість кубітів: 65. Температура роботи: -273°C (близько до абсолютного нуля).

1.2. Google Sycamore (2019, США). Призначення: дослідження нових алгоритмів квантових обчислень. Розробник: Google AI Quantum. Кількість кубітів: 53. Головне досягнення: продемонстровано квантову перевагу – виконано розрахунок за 200 секунд, який класичний суперкомп’ютер виконував би 10 000 років.

1.3. D-Wave Advantage (2020, Канада). Призначення: оптимізаційні задачі, фінансове моделювання, машинне навчання. Розробник: D-Wave Systems. Кількість кубітів: 5000 (адіабатична квантова система).

1.4. Zuchongzhi (2021, Китай). Призначення: високопродуктивні розрахунки, дослідження у квантовій фізиці. Розробник: Університет науки і технологій Китаю (USTC). Кількість кубітів: 66.

2 Нейроморфні комп’ютери (імітація роботи мозку)

2.1. IBM TrueNorth (2014, США). Призначення: моделювання штучного інтелекту, розпізнавання образів. Розробник: IBM. Кількість штучних нейронів: 1 000 000. Особливості: використовує імпульсну (спайкову) нейромережу, як людський мозок.

2.2. Intel Loihi (2018, США). Призначення: розпізнавання зображень, автономна робототехніка, AI-алгоритми. Розробник: Intel. Кількість штучних нейронів: 128 000. Споживана потужність: у десятки разів менше, ніж традиційні процесори.

2.3. SpiNNaker (2018, Велика Британія). Призначення: симуляція мозкової активності, біомедичні дослідження. Розробник: Манчестерський університет. Кількість ядер: 1 000 000 ядер, що імітують нейрони мозку.

3 Оптичні та фотонні комп’ютери

3.1. Optalysys Optical Processor (2020, Великобританія). Розробник: Optalysys. Технологія: обчислення за допомогою лазерних променів і оптичних мікросхем. Призначення: обробка зображень, біоінформатика, моделювання погоди.

3.2. Lightmatter Envise (2022, США). Розробник: Lightmatter. Технологія: використовує фотонні сигнали замість електронних для обчислень. Переваги: енергоспоживання на 90% менше порівняно з традиційними чипами. Призначення: AI, машинне навчання, обробка великих даних.

4 Біокомп’ютери (ДНК-обчислення, молекулярні системи)

4.1. ДНК-комп’ютер Adleman (1994, США). Розробник: Леонард Адлеман (батько ДНК-обчислень). Технологія: ДНК-молекули використовуються для виконання складних математичних операцій. Призначення: криптографія, аналіз великих даних.

4.2. MAYA-II (2004, США). Розробник: Колумбійський університет. Технологія: використовує молекули ДНК як носії інформації. Призначення: біоінформатика, фармакологія, медичні дослідження.

4.3 SynBio DNA Computer (2030, прогноз). Розробник: MIT, Harvard (дослідження тривають). Технологія: використання синтетичної ДНК як інформаційного носія. Призначення: медичні дослідження, біохакінг, персоналізована медицина.

5 Гібридні (універсальні) обчислювальні системи

5.1. Cerebras Wafer-Scale Engine (2019, США). Розробник: Cerebras Systems. Технологія: найбільший у світі чип – 56 разів більший за стандартний CPU. Призначення: AI, машинне навчання, біоінформатика.

5.2. Fugaku (2020, Японія). Розробник: RIKEN, Fujitsu. Технологія: суперкомп’ютер із процесорами ARM. Потужність: понад 442 петафлопси (найпотужніший суперкомп’ютер у 2021 році). Призначення: дослідження клімату, медичні симуляції, AI.

Переваги та перспективи:

– Квантові обчислення можуть експоненційно прискорювати складні задачі.

– Нейроморфні процесори споживають у сотні разів менше енергії.

– Фотонні обчислення забезпечують блискавичну швидкість обробки даних.

– Штучний інтелект набув широкого застосування у всіх сферах.

Проблеми та виклики нових типів обчислювальних машин

– Квантові комп’ютери вимагають екстремально низьких температур.

– Кубіти нестабільні, вимагають корекції помилок.

– Нейроморфні процесори ще не готові до масового використання.

– Фотонні обчислення перебувають на стадії досліджень.

Хоча всі ці технології ще розвиваються, вже зараз виникають нові виклики, такі як необхідність масового впровадження квантових комп’ютерів, потреба в інтеграції нейроморфних процесорів у традиційні комп’ютери, використання біологічних обчислень (ДНК-комп’ютери, молекулярні системи) та розробка стабільних фотонних обчислювальних систем. Саме ці напрямки визначать наступні покоління обчислювальних машин, які буде базуватися на біологічних, квантових та нанотехнологіях, а і використання таких обчислювальних машин може стати новим етапом еволюції та докорінно змінити людське суспільство.

Отже, в ході розвитку обчислювальної техніки та інформаційних технологій клас задач, що реалізовуються обчислювальними машинами та комп’ютерами, істотно розширився. Сучасні комп’ютери вирішують не лише математичні задачі, а й широкий клас завдань «необчислювального» характеру: обробку текстів, графіки та зображень, стиснення та перетворення інформації, розпізнавання образів та текстових конструкцій, інформаційно-пошукові завдання тощо.

4.2. Принципи архітектури фон Неймана.

Будова та функції основних компонентів архітектури фон Неймана

Архітектура фон Неймана (Прінстонська архітектура), запропонована в 1945 році Джоном фон Нейманом [1], залишається основою більшості сучасних комп’ютерних систем. Її ключовою особливістю є єдиний простір пам’яті, де зберігаються як дані, так і програмні інструкції. Ця концепція заклала основу для створення універсальних комп’ютерів, здатних виконувати різні завдання шляхом зміни програмного забезпечення.

Система фон Неймана (рис. 4.9) складається з трьох основних компонентів:

1. Процесор (центральний процесорний пристрій, CPU) – виконує обчислення та керує роботою комп’ютера.

2. Пам’ять – зберігає дані та інструкції програм.

3. Пристрої введення/виведення (I/O) – забезпечують взаємодію комп’ютера з користувачем і зовнішнім світом.

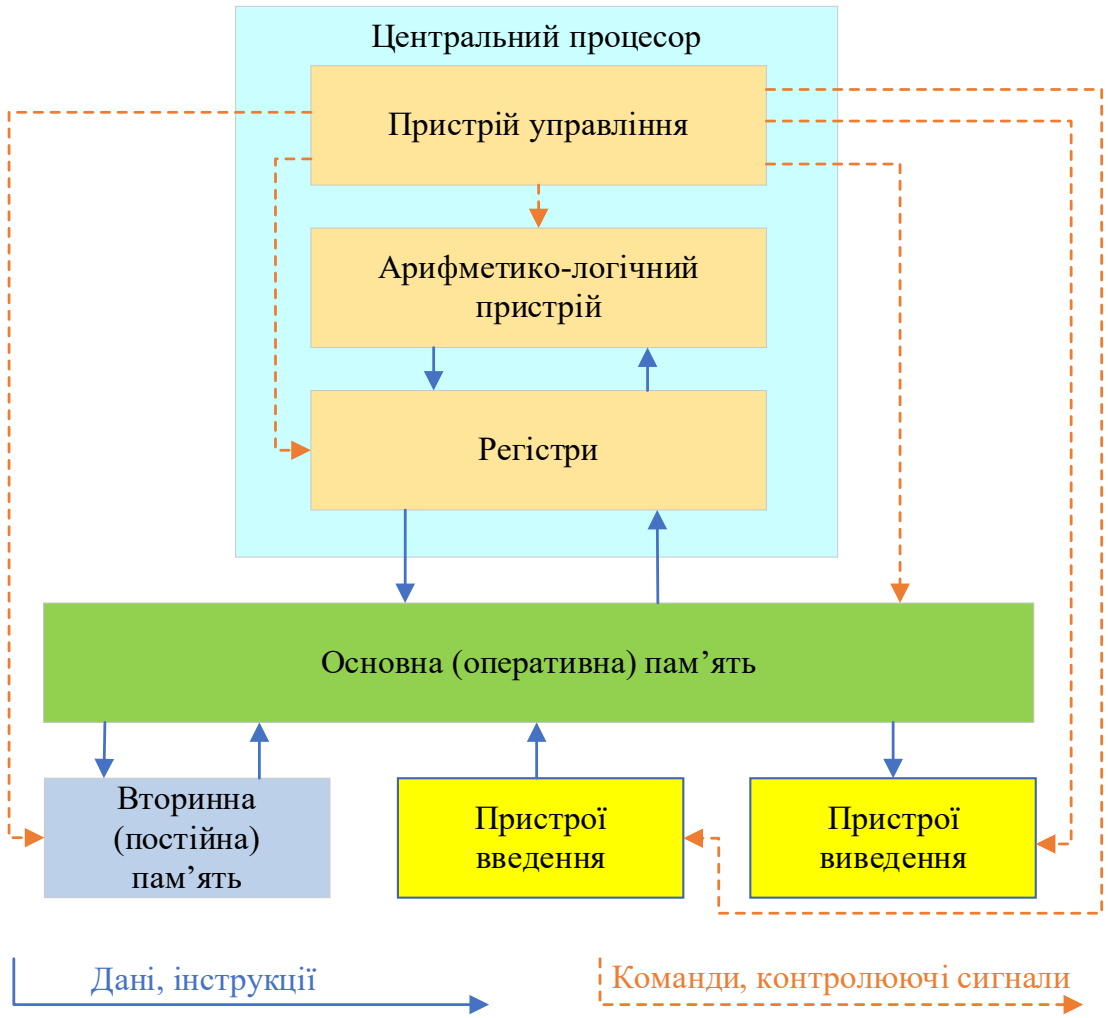

Рисунок 4.9 – Архітектура фон Неймана

Всі ці компоненти взаємодіють між собою через системну шину, яка передає команди, дані та керуючі сигнали. Розглянемо їх більш детально:

Центральний процесор CPU, Central Processing Unit) – це головний обчислювальний елемент комп’ютера, який інтерпретує та виконує інструкції програми. Він складається з таких основних частин:

1. Пристрій управління (Control Unit). Інтерпретує інструкції програми, керує всіма операціями процесора, генерує сигнали синхронізації для всіх компонентів комп’ютера. Контролює рух даних між процесором, пам’яттю та пристроями введення/виведення.

2. Арифметико-логічний пристрій (ALU, Arithmetic Logic Unit). Виконує арифметичні (додавання, віднімання, множення, ділення) та логічні (AND, OR, NOT, XOR) операції, а також відповідає за виконання всіх обчислень у процесорі.

3. Регістри процесора – це основні блоки найбільш швидкодіючої пам’яті, розміщені в самому процесорі, що мають надзвичайно малий обсяг оскільки знаходяться всередині процесора для швидкого доступу до даних та команд. Основні регістри:

– Регістри загального призначення (AX, BX, CX, DX) – зберігають проміжні результати обчислень.

– Лічильник команд (Program Counter, PC) – вказує адресу наступної інструкції.

– Регістр команд (Instruction Register, IR) – містить поточну команду, яку виконує процесор.

– Регістр стану (Flags Register) – відображає поточний стан процесора (результати операцій, умови виконання команд).

4. Пам’ять у комп’ютерах фон Неймана використовується для зберігання як даних, так і програмних інструкцій. Вона організована у вигляді послідовно пронумерованих комірок, доступ до яких здійснюється за унікальними адресами.

5. Пристрої введення/виведення (I/O) – забезпечують отримання даних ззовні, передачу їх у пам’ять та виведення результатів обчислень. Основними пристроями введення є: клавіатура, миша – для введення тексту та команд; сканери, камери – для отримання графічної інформації; датчики та мікрофони – для реєстрації зовнішніх змін і звуків. Основними пристроями виведення є: монітор – для виведення графічної інформації; принтер – для друку тексту та зображень; динаміки – для відтворення звуків.

Для управління системою введення/виведення використовується контролер вводу/виводу (I/O Controller), що керує передачею даних між периферійними пристроями та пам’яттю. Обмін даними також здійснюється через системну шину, яка має обмежену пропускну здатність.

Отже, архітектура фон Неймана визначає принципи побудови сучасних комп’ютерних систем. Вона базується на використанні єдиної пам’яті для

програм і даних, що спрощує розробку програмного забезпечення. Основні компоненти – процесор, пам’ять, пристрої введення/виведення – взаємодіють через системну шину, виконуючи мільйони операцій за секунду, що створює певні обмеження такої архітектури, так звану проблему "вузького місця" пам’яті.

Обмеження архітектури фон Неймана.

Архітектура фон Неймана, хоч і залишається основою сучасних комп’ютерних систем, має кілька суттєвих обмежень, які впливають на продуктивність обчислень. Однією з найбільших проблем є обмежена пропускна здатність пам’яті та шини, що створює так зване "вузьке місце" (bottleneck).

Суть проблеми полягає в наступному: в архітектурі фон Неймана процесор, пам’ять та пристрої введення/виведення використовують спільну шину для передачі даних. Шина – це фізичний канал, через який передаються дані між компонентами комп’ютера (процесором, пам’яттю, накопичувачами, відеокартою тощо). Причини обмежень шини полягають в обмеженні пропускної здатності (Bandwidth): шина має фіксовану кількість бітів, які можна передати за один такт (наприклад, 32 або 64 біти) і при збільшенні обсягу даних шина стає перевантаженою, що призводить до затримок, відповідно виникає певна конкуренція за доступ до шини, процесор, пам’ять, відеокарта, накопичувачі – всі ці компоненти використовують спільну шину, і чим більше пристроїв працює одночасно, тим більше навантаження на шину і тим повільніше передаються дані, так виникають проблеми із затримками (Latency), час очікування на відповідь від шини (затримка) може бути більшим, ніж час виконання команди в процесорі, що особливо помітно при роботі з високошвидкісними пристроями, наприклад, NVMe, SSD або потужними GPU. Так, оскільки інструкції та дані передаються послідовно, а не паралельно, це значно обмежує швидкість роботи системи, наприклад процесор, який працює набагато швидше, ніж пам’ять, часто змушений простоювати в очікуванні на отримання необхідних даних або інструкцій, що є неефективним використанням процесорного часу.

Відповідно, наслідками перевантаження шини є сповільнення обміну даними між процесором і пам’яттю; низька ефективність роботи багатоядерних процесорів, коли всі ядра конкурують за спільний канал обміну даними; обмежена швидкість зовнішніх пристроїв (жорстких дисків, відеокарт, мережевих адаптерів) через недостатню пропускну здатність інтерфейсу.

Розглянемо, які існують методи подолання обмежень архітектури фон Неймана:

– Використання кеш-пам’яті. Кеш-пам’ять (L1, L2, L3) зберігає часто використовувані дані безпосередньо в процесорі, що зменшує необхідність звернення до основної пам’яті.

– Використання багатоканальної пам’яті (Dual-Channel, Quad-Channel). У сучасних системах пам’ять може працювати в кількох каналах одночасно, що

підвищує пропускну здатність, наприклад, двоканальний (Dual-Channel) режим дає подвійне збільшення швидкості передачі даних між процесором і RAM.

– Використання окремих шин для різних компонентів, наприклад, сучасні комп’ютери використовують PCI Express (PCIe) для підключення відеокарт та накопичувачів, що зменшує навантаження на основну шину. Відповідно використання окремих каналів для обміну даними значно покращує продуктивність.

– Використання альтернативних архітектур: Гарвардська архітектура (відокремлені шини для команд і даних), багатоядерні процесори з незалежними кешами для кожного ядра, гетерогенні архітектури (CPU + GPU), де частина задач виконується на спеціалізованих процесорах.

4.3. Основні компоненти сучасної комп'ютерної техніки.

Архітектура персонального комп'ютера.

Персональний комп’ютер є одним із найпоширеніших типів обчислювальних систем, який використовується для широкого спектра завдань: від роботи з документами до складних наукових обчислень та комп’ютерних ігор. Його архітектура базується на принципах фон-нейманівської моделі, але зазнала значних змін завдяки появі багатоядерних процесорів, швидкісної пам’яті, спеціалізованих прискорювачів та розширених можливостей введення/виведення.

Архітектура персонального комп’ютера передбачає розподіл функцій між основними апаратними компонентами: центральний процесор (CPU) – виконує обчислення та керує всіма процесами; оперативна пам’ять (RAM) – тимчасово зберігає програми та дані; системна плата (материнська плата, motherboard) – забезпечує з’єднання всіх компонентів; графічний процесор (GPU) – відповідає за обробку графіки та відео; пристрої зберігання даних (HDD, SSD, NVMe) – забезпечують довготривале збереження інформації; системна шина – передає дані між компонентами; блок живлення (PSU) – забезпечує електроживлення всіх елементів ПК; пристрої введення/виведення (монітор, клавіатура, миша, принтер тощо) – взаємодія користувача з системою. Персональні комп’ютери можуть мати різну форму та конфігурацію – від класичних настільних ПК (desktop) до ноутбуків, ультрабуків, робочих станцій (workstation) та міні-ПК.

Розглянемо більш детально основні компоненти сучасного персонального комп’ютера.

1. Центральний процесор. Центральний процесор (Central Processing Unit, CPU) є "мозком" комп’ютера, що виконує команди програм і керує всіма обчисленнями. Основні характеристики CPU:

- Архітектура: CISC (Intel, AMD) або RISC (ARM, Apple M-серія).
- Кількість ядер: Одноядерні, двоядерні, багатоядерні (4, 6, 8, 16, 64+).
- Тактова частота: Вимірюється в ГГц (часто змінна завдяки турборежиму).

- Кеш-пам’ять: Рівні L1, L2, L3 для швидкого доступу до даних.

- Розрядність: 32-бітні та 64-бітні архітектури.

- Підтримка багатопотоковості: Hyper-Threading (Intel), SMT (AMD).

Сучасні процесори також можуть містити:

- Інтегрований графічний адаптер (iGPU).

- Модулі штучного інтелекту (NPU, AI-ядра).

- Апаратне прискорення криптографії та віртуалізації.

2. Оперативна пам’ять. Оперативна пам’ять (Random Access Memory, RAM) забезпечує тимчасове зберігання даних, необхідних для роботи процесора та програм. Основні типи пам’яті, що використовуються:

- DDR4, DDR5 – сучасні стандарти оперативної пам’яті.

- LPDDR – використовується в ноутбуках та мобільних пристроях.

- HBM – високошвидкісна пам’ять для GPU та серверних систем.

Основними параметрами RAM є:

- Об’єм (4 ГБ – 128 ГБ і більше для серверів).

- Частота (3200 МГц, 4800 МГц, 6000+ МГц).

- Затримки (CL, CAS Latency) – впливає на швидкість доступу до даних.

3. Материнська плата (Motherboard). Материнська плата – це основна плата комп’ютера, яка з’єднує всі компоненти. Основні елементи материнської плати:

- Чипсет – керує роботою процесора, пам’яті та периферійних пристроїв.

- Слоти для RAM (DIMM) – дозволяють встановлювати модулі оперативної пам’яті.

- Роз’єми PCIe – для відеокарт, SSD, мережевих адаптерів тощо.

- Порти SATA, M.2 – для підключення накопичувачів.

- USB, аудіовиходи, мережеві інтерфейси (LAN, Wi-Fi, Bluetooth).

4. Графічний процесор. Графічний процесор (Graphics Processing Unit, GPU) відповідає за рендеринг зображень, відео та обчислення, пов’язані зі штучним інтелектом. Основними типами GPU є:

- Інтегровані (iGPU) – вбудовані в процесор, використовують оперативну пам’ять.

- Дискретні (NVIDIA, AMD, Intel ARC) – окремі потужні відеокарти.

Сучасні відеокарти підтримують наступні технології:

- Ray Tracing (апаратне трасування променів).

- GPGPU (загальні обчислення на GPU, наприклад, для AI, наукових задач).

5. Пристрої зберігання даних (HDD, SSD, NVMe). Зберігання даних забезпечується жорсткими дисками та твердотільними накопичувачами. Основними типами накопичувачів є:

- HDD (жорсткий диск) – дешевший, але повільніший ніж інші, і є чутливим до фізичного впливу – ударів чи вібрації.

- SSD (твердотільний диск) – швидший за HDD, набагато стійкіший до фізичних впливів, але має невелику кількість циклів перезапису і може деградувати при тривалих високих температурах.

- NVMe SSD (PCIe 4.0, 5.0) – надшвидкісні накопичувачі для продуктивних систем. За терміном служки аналогічний до звичайного SSD, але може швидше досягати межі зносу за рахунок інтенсивного використання.

Ключові параметри пристроїв зберігання:

- Об’єм (256 ГБ – 8 ТБ і більше).

- Швидкість читання/запису (до 7000+ МБ/с для NVMe).

6. Блок живлення (PSU) – відповідає за стабільне електроживлення всіх компонентів. Стандартна потужність: від 300 Вт (офісні ПК) до 1200+ Вт (ігрові та серверні системи).

7. Системи охолодження:

- Повітряне охолодження (кулери, радіатори).

- Рідинне охолодження (AIO, кастомні системи).

- Теплопровідні матеріали (термопаста, теплові трубки).

Окремо варто зупинитись на організації материнської плати. Розглянемо її складові за класичною схемою та відповідно до сучасних підходів.

У класичній архітектурі, до середини 2000-х, материнська плата мала два основні чипи:

1. Північний міст (Northbridge) відповідав за швидкісні компоненти:

- Процесор (CPU) – з’єднання через системну шину (FSB).
- Оперативна пам'ять (RAM) – контролер пам’яті знаходився тут (до ери вбудованих контролерів у CPU).
- Графічний інтерфейс (AGP, PCI Express x16).
- Шина до південного мосту.

2. Південний міст (Southbridge) обслуговував повільніші пристрої та периферію:

- SATA / IDE контролери (жорсткі диски, SSD).
- USB, FireWire.
- PCI, PCIe x1.
- BIOS/UEFI.
- Аудіо-кодек.
- Мережевий контролер.
- Клавіатура, миша, порти COM, LPT.

З середини 2000-х (архітектури Intel Nehalem, AMD Athlon 64), почало відбуватись об’єднання у єдиний чип: контролер пам’яті і часто PCI Express були інтегровані в сам процесор, усі функції південного мосту та частина північного

були об’єднані у Platform Controller Hub (PCH) в Intel або Fusion Controller Hub (FCH) в AMD, а північний міст зник як окремий чип. Отже, сучасна схема виглядає так:

1. CPU містить:

- Контролер пам’яті (DDR4, DDR5).
- Лінії PCI Express (для відеокарти і NVMe SSD).
- Інтегроване графічне ядро (в деяких моделях).

2. PCH або FCH містять:

- SATA, USB, Ethernet, аудіо.
- Додаткові PCIe лінії для розширення.
- SPI/SMBus, прошивка BIOS/UEFI.

Окремо виділимо таку схему організації архітектури комп’ютера, як розміщення системи на одній платі (System-on-Chip). Так у мобільних комп'ютерах (ноутбуки, ARM-чипи, Apple M1/M2) майже все інтегровано в один кристал: CPU, GPU, контролери пам’яті, I/O, а окремий "південний міст" як чип вже практично відсутній, але Platform Controller Hub ще використовується у настільних комп'ютерах Intel (серії Z, B, H).

Архітектура System-on-Chip (SoC) – це концепція, за якої всі основні компоненти комп’ютерної системи інтегруються на одному кристалі кремнію. На відміну від традиційної материнської плати, де процесор, контролери, пам’ять і периферійні інтерфейси розміщені окремо, SoC поєднує їх у єдину мікросхему, що суттєво зменшує розміри, енергоспоживання і затримки між модулями.

У типовій архітектурі SoC центральну роль відіграє CPU – кілька ядер загального призначення (наприклад, ARM Cortex-A або RISC-V), які виконують основні обчислення та керування системою. Поряд із ним розміщено GPU – графічний процесор, який відповідає за візуалізацію, прискорення обчислень із плаваючою комою та обробку даних для штучного інтелекту. На одному кристалі також інтегруються спеціалізовані прискорювачі: NPU (Neural Processing Unit) для нейромережевих обчислень, DSP (Digital Signal Processor) для обробки сигналів, та іноді ISP (Image Signal Processor) – для роботи з камерами.

Важливу частину SoC становить підсистема пам’яті. Безпосередньо на кристалі можуть розташовуватись кеші L1–L3, регістри, буфери, а також контролери зовнішньої оперативної пам’яті (DRAM). Ці контролери з’єднані з процесорними ядрами через внутрішню шину або мережу на кристалі (NoC, Network-on-Chip), яка забезпечує високошвидкісну комунікацію між усіма компонентами. Архітектура NoC замінює класичні послідовні шини, дозволяючи паралельну передачу даних між численними блоками без конфліктів.

Інтегровані контролери введення/виведення забезпечують підтримку таких інтерфейсів, як USB, PCI Express, SATA, UART, SPI, I²C, Ethernet тощо. У

мобільних і вбудованих системах SoC часто містить радіомодулі (Wi-Fi, Bluetooth, LTE, 5G, GPS), аудіокодеки, енергоконтролери та систему керування живленням (PMU). Це дозволяє створювати повноцінні комп’ютерні платформи – від смартфонів і планшетів до автомобільних контролерів та мікросупутників – без потреби в додаткових мікросхемах.

Ключова перевага SoC полягає у щільній інтеграції та оптимізації взаємодії компонентів. Завдяки коротким з’єднанням зменшуються затримки, тепловиділення й споживання енергії, підвищується швидкість обміну даними. Проте така інтеграція має й недоліки: модернізація окремих частин (наприклад, заміна CPU або GPU) неможлива без повної заміни кристала; також ускладнюється тепловідведення і тестування, збільшується імовірність браку.

Сучасні SoC (наприклад, Apple M-серії, Qualcomm Snapdragon, AMD Ryzen APU, NVIDIA Orin) демонструють перехід від класичної архітектури “процесор + чипсет” до єдиної обчислювальної платформи, де всі елементи – обчислювальні, графічні, комунікаційні й керуючі – працюють у тісній апаратній синергії.

Сучасні мобільні пристрої та їх архітектура.

Мобільні пристрої – це компактні, портативні обчислювальні системи, які включають смартфони, планшети, смарт-годинники, електронні книги та інші гаджети, що забезпечують безперервний зв’язок, мультимедійні можливості та мобільні обчислення.

На відміну від традиційних персональних комп’ютерів, мобільні пристрої мають енергоефективну архітектуру, високу інтеграцію компонентів та оптимізацію для роботи в автономному режимі. Вони зазвичай базуються на ARM-архітектурі, яка забезпечує низьке енергоспоживання, високу продуктивність і компактні розміри чипів. Так сучасні мобільні пристрої поєднують у собі потужний процесор, оперативну пам’ять, дисплей, накопичувачі, бездротові модулі зв’язку та елементи керування (різноманітні сенсори, камери, мікрофони, тактильний зворотний зв’язок тощо).

Архітектура мобільних пристроїв суттєво відрізняється від традиційних ПК через необхідність оптимізації енергоспоживання та високої інтеграції компонентів. Вона базується на згаданій раніше системі-на-кристалі SoC яка об’єднує всі ключові компоненти в одному чипі. Так основними компонентами мобільного пристрою є: центральний процесор для виконання основних обчислень, графічний процесор для обробки зображень, відео, 3D-графіки, оперативна пам’ять – інтегрована або як окремий модуль, блоки обробки штучного інтелекту для прискорення AI-обчислень, сигнальний процесор для обробки звуку та відео, контролери бездротового зв’язку (Wi-Fi, Bluetooth, 5G, NFC) та інтегральну схему управління живленням, що використовується для оптимізації та управління живленням пристрою та всіх його компонентів.

Найбільш популярними на сьогодні SoC-платформами для мобільних пристроїв є: Qualcomm Snapdragon, MediaTek Dimensity та Samsung Exynos для Android-смартфонів, Apple A-серія для пристроїв лінійок iPhone, iPad, та Google Tensor для використання штучного інтелекту в смартфонах Pixel.

Розглянемо більш детально характеристики компонентів мобільних обчислювальних систем.

1. Процесор мобільного пристрою:

- ARM-архітектура – переважає в мобільних пристроях завдяки низькому енергоспоживанню.

- Багатоядерність – мобільні процесори мають 4-12 ядер для ефективного розподілу навантаження.

- Динамічна частота (big.LITTLE) – поєднання потужних ядер для складних задач, та енергоефективних для фонових процесів.

2. Пам’ять і сховища даних:

- Оперативна пам’ять: LPDDR5 – енергоефективна та має високу швидкість доступу.

- Постійна пам’ять: вбудовані накопичувачі (UFS 3.1, UFS 4.0), що значно швидші за традиційні SSD та картки microSD.

- Кеш-пам’ять різних рівнів (L1, L2, L3), що зменшує навантаження на основну пам’ять.

3. Дисплеї. Основними типами, що використовуються на сьогодні є:

- LCD (IPS) – бюджетні рішення.

- OLED, AMOLED – яскраві кольори, глибокий чорний, енергоефективність.

- LTPO OLED – адаптивна частота оновлення для економії заряду.

4. Серед графічних процесорів варто виділити Adreno (Qualcomm), Mali (ARM), Apple GPU, Xclipse (Samsung). Вони використовуються для графіки, відео, доповненої реальності (AR) та AI-обчислень тощо.

4. Бездротові технології зв’язку

- Мобільні мережі: 4G LTE, 5G – забезпечують високу швидкість передачі мобільних даних.

- Wi-Fi – швидкісне підключення до Інтернету.

- Bluetooth – енергоефективний бездротовий зв’язок.

- NFC (Near Field Communication) – безконтактні датчики, зокрема використовується для систем безконтактної оплати.

5. Сенсори та додаткові компоненти

- Біометричні датчики: відбитки пальців, Face ID.

- Гіроскоп, акселерометр, компас – визначення положення пристрою.

- Датчик освітлення – регулювання яскравості дисплея.

- Барометр, термометр, пульсометр – у смарт-годинниках та фітнес-браслетах.

6. Живлення та система охолодження

Акумулятори – літій-іонні (Li-Ion), літій-полімерні (Li-Po), ємністю 3000–7000 мА·год, з функціями швидкої зарядки (Fast Charge, Quick Charge, PD).

Охолодження:

- Пасивне (без вентиляторів) – оптимізовані алгоритми управління живленням.

- Випарні камери – використовуються у геймерських смартфонах для відведення тепла.

Спеціалізовані та IoT-системи: мікроконтролери, інтегровані (вбудовані) системи.

Сучасні обчислювальні пристрої охоплюють не лише персональні комп’ютери та мобільні гаджети, а й спеціалізовані системи та системи інтернету речей (Internet of Things – IoT), що забезпечують керування технікою, автоматизацію процесів та обробку даних у реальному часі. До таких систем належать мікроконтролери та вбудовані системи (embedded systems), які широко використовуються в автомобільній промисловості, робототехніці, побутовій техніці, медичному обладнанні, аерокосмічній техніці та промисловій автоматизації. Вони мають специфічну архітектуру, оптимізовану для виконання певних завдань при мінімальному енергоспоживанні та високій надійності.

Мікроконтролер (MCU, Microcontroller Unit) – це мікросхема, яка поєднує процесор, пам’ять і периферійні інтерфейси на одному чипі. Його головне призначення – керування пристроями в реальному часі за допомогою заздалегідь запрограмованих алгоритмів.

Серед особливостей мікроконтролерів слід виділити: низьке енергоспоживання (від кількох мікроват до кількох ват), виконання специфічних завдань (наприклад, керування двигуном, обробка даних з датчиків), наявність вбудованої пам’яті (ROM, RAM, EEPROM) для збереження програм і даних та інтерфейсів типу SPI, I^2C, UART, GPIO для підключення різноманітних датчиків і пристроїв.

Основними компонентами MCU є:

- Центральний процесор (CPU) – 8, 16 або 32-бітний RISC/CISC-ядро.

- ОЗП (RAM) – тимчасове збереження даних під час виконання програм.

- ПЗП (ROM, Flash) – збереження прошивки пристрою.

- Цифрові та аналогові входи/виходи (GPIO, ADC, DAC) – призначені для керування пристроями та зчитування даних.

- Інтерфейси зв’язку (UART, SPI, I²C, CAN, USB, Ethernet) – використовуються для обміну даними з іншими пристроями.

- Таймери та лічильники, що застосовуються для точного вимірювання часу та керування процесами.

Популярними на сьогодні мікроконтролерами є:

- 8-бітні (ATmega328, PIC16F877A) – використовуються в простих пристроях (Arduino, сенсорні системи).
- 16-бітні (MSP430, PIC24) – більш продуктивні, застосовуються у вбудованих системах, промислових контролерах.
- 32-бітні (STM32, ESP32, ARM Cortex-M) – потужні, підтримують бездротові комунікації, багатозадачність, штучний інтелект.

Основними напрямками застосування мікроконтролерів є різноманітні види техніки: побутова та IoT-системи (керування пральними машинами, мікрохвильовками, кондиціонерами, датчики температури, розумні лампи, системи безпеки), автомобільна (ABS, подушки безпеки, круїз-контроль, електропідсилювач керма тощо), промислова (роботи, PLC-контролери, автоматизовані виробничі лінії), та медична (кардіостимулятори, глюкометри, апарати ШВЛ тощо).

На базі мікроконтролерів можуть будуватись цілі вбудовані системи.

Вбудована система (Embedded System) – це спеціалізований комп’ютер, вбудований у певний пристрій для виконання конкретного завдання. Така система поєднує апаратне забезпечення та програмне керування для автоматизації та управління пристроями, так такі системи: виконують чітко визначені функції (керування автомобілем, розумний термостат), оптимізовані під конкретні завдання, реалізовані на базі мікроконтролерів, SoC або FPGA, та мають реальний або квазі-реальний час обробки (Real-Time Operating System, RTOS).

Вбудовані системи можна класифікувати за різними ознаками:

За рівнем складності:

- Прості однозадачні системи (stand-alone systems) – виконують одну задачу (термометри, електронні годинники).

- Складні мережеві системи (networked systems) – обмінюються даними з іншими пристроями (IoT, мережеві контролери).

- Критично важливі системи реального часу (real-time systems, RTS) – повинні працювати без збоїв у заданий момент (медична техніка, авіоніка).

За апаратною основою:

- На базі мікроконтролерів (MCU-based) – малопотужні пристрої з жорсткою логікою роботи.

- На базі систем-на-кристалі (SoC-based, наприклад, Raspberry Pi, NVIDIA Jetson) – потужні, можуть виконувати складні обчислення.

- На базі FPGA (програмовані логічні схеми) – налаштовуються під специфічні завдання (обробка сигналів, фінансові обчислення).

Отже, мікроконтролери та вбудовані системи є ключовими компонентами сучасних технологій, оскільки вони забезпечують автоматизацію, зв’язок та інтелектуальне керування пристроями. Розвиток IoT, AI та автономних систем стимулює появу більш потужних вбудованих платформ, що використовують нейромережі, енергоефективні SoC та високошвидкісні бездротові інтерфейси. Майбутнє цих систем – в інтеграції AI, машинного навчання та хмарних технологій для створення інтелектуальних пристроїв нового покоління.

Мейнфрейми та суперкомп’ютери

Мейнфрейми та суперкомп’ютери – це високопродуктивні обчислювальні системи, що використовуються для вирішення складних завдань у бізнесі, науці, фінансах, медицині та військовій сфері.

Хоча обидва типи систем мають високу продуктивність, їхні архітектурні особливості та сфери застосування суттєво відрізняються.

Мейнфрейми (mainframes) – великі серверні системи, що забезпечують масштабованість, надійність та використовуються у банківських системах, державних органах, великих корпораціях та дата-центрах для централізованої обробки великих обсягів даних та забезпечення безперервної роботи критично важливих сервісів.

Основні характеристики мейнфреймів:

- Висока надійність – час безперервної роботи – 99.999% uptime;
- Масштабованість – підтримка десятків тисяч користувачів одночасно;
- Підтримка великої кількості операційних систем (Linux, z/OS, UNIX);
- Мультипроцесорність та паралельна обробка запитів;
- Робота з великими обсягами транзакцій у фінансовій сфері.

Мейнфрейми використовують спеціалізовану архітектуру, оптимізовану для надійності та багатопотокової обробки даних. До основних технічних характеристик мейнфреймів можна віднести:

- Центральні процесори (CPUs) – багатоядерні, можуть мати до 100+ ядер;

- Системна пам’ять (RAM) – від 512 ГБ до десятків ТБ;

- Високошвидкісні накопичувачі – використовують NVMe SSD, RAID-масиви;

- Інтерфейси вводу/виводу – PCIe, Fibre Channel, InfiniBand для підключення дисків та мереж;

- Спеціалізовані криптографічні модулі – для захисту даних та безпечних транзакцій.

Основними представниками мейнфреймів на сьогодні є наступні:

- IBM Z-series (IBM z16, z15, z14) – найпоширеніші корпоративні мейнфрейми, що використовуються у банках та державних установах.
- Fujitsu GS21 – японські мейнфрейми для обробки фінансових операцій.
- Hitachi AP8000 – масштабовані мейнфрейми для дата-центрів.

Мейнфрейми зазвичай використовуються для вирішення спеціальних задач у наступних сферах:

- Банківська сфера – обробка мільйонів транзакцій у реальному часі;
- Страхові компанії – управління клієнтськими базами та розрахунок ризиків;
- Урядові структури – зберігання та аналіз державних даних;
- Телекомунікації – управління інфраструктурою мобільних операторів;
- Авіація – облік та управління бронюванням авіаквитків.

Суперкомп’ютери (supercomputers) – найпотужніші обчислювальні системи, що здатні виконувати квадрильйони операцій на секунду (ExaFLOPS), та які використовуються для вирішення завдань, які вимагають таких гігантських обчислювальних ресурсів – складних наукових та інженерних розрахунків, таких як моделювання кліматичних змін, розробка ліків, квантова фізика та космічні дослідження.

Основні характеристики суперкомп’ютерів:

- Паралельна обробка даних – тисячі процесорів та графічних прискорювачів;
- Висока продуктивність – терафлопси (TFLOPS) і екзафлопси (EFLOPS);
- Оптимізовані для наукових розрахунків та машинного навчання;
- Працюють на Linux-базованих операційних системах.

Сучасні суперкомп’ютери базуються на кластерній архітектурі та використовують десятки тисяч процесорів і графічних прискорювачів. Основними компонентами, що використовуються для суперкомп’ютерів є: процесори AMD EPYC, Intel Xeon, ARM-чипи; графічні прискорювачі NVIDIA H100, AMD Instinct, Google TPU; оперативна пам’ять від 100 ТБ до кількох петабайтів; Високошвидкісні інтерконекти InfiniBand, NVLink, OmniPath для швидкого обміну даними та дискові масиви на базі Lustre, NVMe SSD для збереження та обробки даних.

Найпотужніші суперкомп’ютери світу постійно відображаються на спеціальному сайті Top 500 (рис. 4.10) [17].

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698

Рисунок 4.10 – Найпотужніші суперкомп’ютери на червень 2025 р.

Суперкомп’ютери використовуються зазвичай для наступних задач:

- Кліматологія – моделювання змін клімату;
- Медицина – симуляція білкових структур, розробка ліків;
- Астрофізика – дослідження чорних дір і космічних явищ;
- Квантова механіка – моделювання складних фізичних процесів;
- Штучний інтелект – тренування великих нейромереж (ChatGPT, AlphaFold).

4.4. Будова сучасної комп'ютерної техніки

Процесор в архітектурі фон-Неймана працює за циклом «вибірка – декодування – виконання», який виконується безперервно, поки не буде завершено виконання програми:

1. Вибірка інструкції (Fetch): Команда зчитується з комірки пам’яті, адреса якої міститься в лічильнику команд (PC), і поміщається в регістр інструкцій (IR). Після цього PC збільшується, щоб вказувати на наступну інструкцію, яка буде зчитана в наступному циклі.

2. Декодування інструкції (Decode): Інструкція в IR інтерпретується блоком декодування, який визначає тип операції (арифметична, логічна, передача даних, керуюча тощо) та які операнди необхідні для виконання.

2.1. Визначення адреси та операндів (Operand Fetch): Якщо інструкція вимагає доступу до пам’яті, процесор визначає ефективну адресу даних. У разі прямої адресації дані одразу зчитуються, при непрямій – спочатку отримується

адреса з пам’яті. Операнди завантажуються в регістри для подальшої обробки. Для регістрових команд цей етап може бути пропущений.

3. Виконання інструкції (Execute): Блок керування генерує сигнали, які активують необхідні вузли процесора. Дані передаються в арифметико-логічний пристрій (АЛП), де виконуються математичні чи логічні операції.

3.1. Збереження результату (Write Back): Результати зберігаються у регістрах або оперативній пам’яті, а при потребі – надсилаються на пристрої введення-виведення. Якщо інструкція змінює порядок виконання (наприклад, умовний перехід), лічильник команд (PC) оновлюється відповідною адресою.

Типи процесорних архітектур.

Сучасна комп’ютерна техніка значною мірою залежить від процесорної архітектури, яка визначає спосіб виконання обчислень, обробку команд, взаємодію з пам’яттю та ефективність виконання програмного забезпечення.

Процесор (Processor) – функціональний блок, який **Інтерпретує** та **Виконує інструкції.** Процесор складається щонайменше з блоку управління інструкціями та арифметико-логічного блоку. Процесори поділяються за типом набору інструкцій (Instruction Set Architecture, ISA), що визначає формат і набір команд, які може виконувати процесор. Наприклад, до набору інструкцій входять операції на зразок додавання чисел, зсувів бітів або порівняння даних. Це свого роду "мова", якою взаємодіють програми з апаратним забезпеченням. ISA також задає формат інструкцій, режим адресації пам’яті та спосіб обміну даними між регістрами процесора і пам’яттю (рис. 4.11).

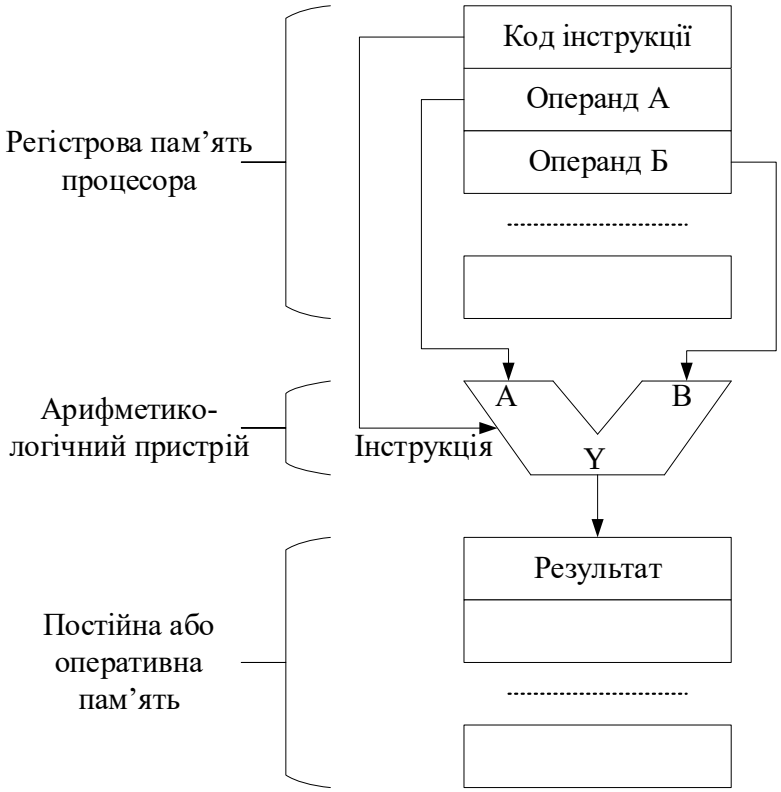

Рисунок 4.11 – Схематичне зображення ISA

Розглянемо типи процесорів:

1. Конвеєрні процесори – сучасні процесори, що використовують конвеєрний принцип обробки інформації, що дозволяють обробляти більше однієї команди одночасно. Цей принцип має на увазі, що в кожний момент часу процесор працює над різними стадіями виконання кількох команд, причому виконання кожною стадії виділяються окремі апаратні ресурси. За черговим тактовим імпульсом кожна команда у конвеєрі просувається на наступну стадію обробки, виконана команда залишає конвеєр, а нова надходить до нього.

Конвеєризація – архітектурний прийом, що використовується в сучасних процесорах з метою підвищення швидкодії.

Ідея полягає у поділі обробки комп'ютерної команди на послідовність незалежних стадій із збереженням результатів наприкінці кожною стадії.

Зазвичай для виконання кожною інструкції потрібно здійснити деяке кількість однотипних операцій, наприклад, отримання інструкції, розкодування інструкції тощо. Кожну з цих операцій зіставляють однієї ланки конвеєра. Деякі сучасні процесори мають більше 30 ступенів у конвеєрі, що підвищує продуктивність процесора, але, однак, призводить до збільшення тривалості простою (наприклад, у випадку помилки у передбаченні умовного переходу). Не існує єдиного думки з приводу оптимальної довжини конвеєра: різні програми можуть мати різні вимоги.

За способом синхронізації роботи ланки конвеєри можуть бути синхронними та асинхронними. Для традиційних ОМ характерні синхронні конвеєри. Пов'язано це перш за все всього, із синхронним характером роботи процесорів. Ланки конвеєрів у процесорі зазвичай розташовуються близько один від одного, завдяки чому тракти поширення сигналів синхронізації виходять досить короткими і фактор «перекосу» сигналів стає не настільки суттєвим. Асинхронні конвеєри виявляються корисними, якщо зв'язок між ланками не такий сильна, а довжина сигнальних трактів між різними ланками може сильно відрізнятись.

Суперконвеєрний процесор – процесор, з апаратно реалізованим режимом конвеєрної обробки та числом ступенів конвеєра понад 8.

Ефективність конвеєра знаходиться у прямій залежно від того, з якою частотою на його вхід подаються об'єкти обробки.

Суперконвеєризація дозволяє покращити продуктивність процесора за рахунок підвищення частоти, з якою команди подаються на конвеєр і переміщаються по ньому.В звичайному конвеєрі ця частота обмежена часом обробки в «найповільнішому» ступені конвеєра.В суперконвеєрі можливість збільшення частоти досягається шляхом виявлення «повільних» ступенів та розбиття їх на кілька простих таким чином, щоб час обробки в кожній з них не перевищувало аналогічного показника для інших сходів конвеєра.

Головна вимога – можливість реалізації операції у кожній ланці конвеєра найбільш простими технічними засобами, а отже, з мінімальними витратами часу.Другою важливою умовою є однаковість затримки на всіх ланках.

Показником для зарахування процесора до суперконвеєрних служить число ланок у конвеєрі команд. До суперконвеєрних відносять процесори, де таких ланок більше шести. Першим серійним суперконвеєрним процесором вважається MIPS R4000, конвеєр команд якого включає в себе вісім ступенів.

Поряд із поняттям суперконвеєризації застосування знайшов і інший термін – «гіперконвеєризація», який компанія Intel використовувала при описі процесора Pentium IV з його конвеєром команд із 20 ступенів.

На жаль, виграш, що досягається за рахунок суперконвеєризації, на практиці може опинитися лише візуальним. Подовження конвеєра веде не тільки до посилення проблем, характерних для будь-якого конвеєра, а й до виникнення додаткових складнощів. В довгому конвеєрі зростає ймовірність конфліктів. Дорожче обходиться помилка передбачення переходу – доводиться очищати більшу кількість ступенів конвеєра, на, що потрібно більше часу. Ускладнюється логіка взаємодії ланок конвеєра. Відображенням цих проблем стало те, що Intel у своїх розробках спочатку збільшила довжину конвеєра до 31 ступеня (ядро Prescoot), а в наступних моделях (процесор Core 2) скоротила число ланок до 14.

Суперскалярний процесор – процесор, здатний паралельно виконувати декілька команд за один такт. Такий режим роботи став можливий завдяки наявності в сучасних процесорах кількох функціональних пристроїв.

Суперскалярним називається центральний процесор (ЦП), який одночасно виконує більше ніж одну скалярну команду. Це досягається за рахунок включення до складу ЦП кількох самостійних функціональних блоків, кожен з яких відповідає за свій клас операцій і може бути присутнім у процесорі в декількох примірниках. Так, у мікропроцесорі Pentium III блоки цілочисленної арифметики та операцій з плаваючою комою дубльовані, а в мікропроцесорах Pentium 4 і Athlon – тройовані.

Суперскалярність припускає паралельну роботу кількох функціональних блоків, що можливо лише за одночасного виконанні кількох скалярні команди. Останнє умова добре поєднується з конвеєрною обробкою, при цьому бажано, щоб таких конвеєрів було кілька, наприклад два чи три. Зрозуміло, у цьому у разі ступінь вибірки команд, загальна для всіх конвеєрів повинна в кожному такті витягувати з пам'яті відразу кілька команд. За цією сходом розташовується блок диспетчеризації, що відповідає за розподіл команд конвеєрами.

Поява цією технології призвело до суттєвого збільшення продуктивності, в той же час існує певний межа зростання числа функціональних пристроїв при перевищенні якого продуктивність практично перестає зростати, а функціональні пристрої простоюють. Частковим рішенням цією проблеми є, наприклад, гіперпотокова технологія.

Класифікація Флінна – класифікація архітектур ЕОМ за ознаками наявності паралелізму в потоках команд та даних (рис. 4.12).

	Одиночний потік інструкцій Single Instruction	Множинний потік інструкцій Multiple Instruction
Одиночний потік даних Single Data	SISD	MISD
Множинний потік даних Multiple Data	SIMD	MIMD

Рисунок 4.12 – Класифікація Флінна

Очевидно, самої ранньої та найбільш відомої є класифікація архітектур обчислювальних систем, запропонована 1966 року М.Флінном. У 1984 році Ванг та Бріггс зробили деякі доповнення до класифікації Флінна, конкретизувавши класи SISD, SIMD, MIMD.

Класифікація базується на понятті потоку, під яким розуміється послідовність елементів, команд або даних, що обробляється процесором. На основі кількості потоків команд та потоків даних Флінн виділяє чотири класи архітектура: SISD, MISD, SIMD, MIMD.

SISD (Single Instruction Stream & Single Data Stream) або ОКОД (Одиничний потік Команд та Одиночний потік Даних) – архітектура комп'ютера, в якій один процесор виконує один потік команд, оперуючи одним потоком даних.

Представниками цього класу є, раніше всього, класичні фон- неймановські ОМ, де є тільки один потік команд, команди обробляються послідовно та кожна команда ініціює одну операцію з одним потоком даних.

SIMD (Single Instruction Stream & Multiple Data Stream) або ОКОД (Одиничний потік Команд та Множинний потік Даних) – архітектура, в якій є можливість виконувати одну арифметичну операцію відразу над багатьма даними – елементами вектора.

Безперечними представниками класу SIMD вважаються матриці процесорів, де єдине керуюче пристрій контролює множина процесорних елементів. Усі процесорні елементи одержують від пристрою управління однакову команду та виконують її над своїми локальними даними. У цей клас

можна включити і векторно-конвеєрні ПС, якщо кожен елемент вектор розглядати як окремий елемент потоку даних.

MISD (Multiple Instruction Stream & Single Data Stream) або МКОД (Множинний потік Команд та Одиночний потік Даних) – тип архітектури, де множина процесорів обробляють один і той же потік даних.

Прикладом такий архітектура могла б служити ОС, на процесори якої подається спотворений сигнал, а кожен з процесорів обробляє цей сигнал за допомогою свого алгоритму фільтрації. Проте ні Флінн, ні інші спеціалісти в області архітектури комп'ютерів досі не змогли представити переконливий приклад реально існуючої обчислювальної системи, побудованої на даному принципі. Ряд дослідників відносять до цього класу конвеєрні системи, однак це не знайшло остаточного визнання. Звідси прийнято вважати, що поки даний клас порожній. Наявність порожнього класу не слід рахувати недоліком класифікації Флінна. Такі класи, на думку деяких дослідників можуть стати надзвичайно корисними для розробки принципово нових концепцій у теорії та практиці побудови обчислювальних систем.

MIMD (Multiple Instruction Stream & Multiple Data Stream) або МКМД (Множинний потік Команд та Множинний потік Даних) – концепція архітектури комп'ютера, що використовується для досягнення паралелізму обчислень. Клас припускає наявність у обчислювальній системі множини пристроїв обробки команд, об'єднаних у єдиний комплекс та працюючих кожне зі своїм потоком команд та даних.

Клас MIMD надзвичайно широкий, оскільки включає в себе всілякі мультипроцесорні системи. Крім того, залучення до класу MIMD залежить від трактування. Так векторно-конвеєрні ОС можна цілком віднести і до класу MIMD, якщо конвеєрну обробку розглядати як виконання множини команд (операцій сходів конвеєра) над множинним скалярним потоком.

Матричні обчислювальні системи – найбільш поширені представники класу SIMD, краще всього пристосовані для вирішення завдань, що характеризуються паралелізмом незалежних даних. Матрична система складається з множини процесорних елементів, що працюють паралельно та оброблювальних свій потік даних.

Призначення матричних обчислювальних систем – обробка великих масивів даних. В основі матричних систем лежить матричний процесор (array processor), що складається з масиву процесорних елементів (ПЕ). системи мають загальний керуючий пристрій, що генерує потік команд, і велика кількість ПЕ, що працюють паралельно та оброблювальних кожен свій потік даних. Однак на практиці, щоб забезпечити достатню ефективність системи при вирішенні широкого кола завдань, необхідно організувати зв'язку між процесорними елементами так, щоб найбільш повно завантажити процесори роботою. Саме

характер зв'язків між ПЕ та визначає різні властивості системи. Подібна схема застосовна і для векторних. обчислень.

Між матричними та векторними системами є суттєва різниця. Матричний процесор інтегрує множину ідентичних функціональних блоків (ФБ), логічно об'єднаних у матрицю і працюючих у SIMD – стилі. Проте, не дуже суттєво, як конструктивно реалізована матриця процесорних елементів – на єдиному кристалі чи кількох. Важливим є сам принцип – ФБ логічно скомпоновані в матрицю і працюють синхронно, тобто присутній лише один потік команд для всіх. Векторний процесор має вбудовані команди для обробки векторів даних, що дозволяє ефективно завантажити конвеєр з функціональних блоків. В свою чергу, векторні процесори простіше використовувати, тому, що команди для обробки векторів – це більше зручна для людини модель програмування, ніж SIMD.

Паралельну обробку множинних елементів даних забезпечує масив процесорних елементів (МПЕ). Єдиний потік команд, керуючий обробкою даних у МПЕ, генерується контролером масиву процесорних елементів (КМП). КМП виконує послідовний програмний код, реалізує операції умовного та безумовного переходів, транслює в МПЕ команди, дані та сигнали управління. Команди обробляються процесорними елементами у режимі жорсткої синхронізації. Сигнали управління використовуються для синхронізації команд і пересилань, а також для керування процесом обчислень, зокрема визначають, які ПЕ масиву повинні виконувати операцію, а які – ні. Команди, дані та сигнали управління передаються з КМП до масиву процесорних елементів по шині широкомовної розсилки. Оскільки виконання операцій умовного переходу залежить від результатів обчислень, результати обробки даних у масиві процесорів транслюються у КМП по шині результату.

Для забезпечення користувача зручним інтерфейсом при створенні та налагодженні програм до складу подібних ОС зазвичай включають фронтальну ОМ (front-end computer). У ролі такої ОМ виступає універсальна обчислювальна машина, на яку додатково покладається завдання завантаження програм та даних у КМП. Крім того, таке завантаження може проводитися і безпосередньо з пристроїв введення / виведення. Після завантаження КМП приступає до виконання програми, транслюючи в МПЕ з широкомовної шині відповідні SIMD- команди.

Розглядаючи масив ПЕ, слід враховувати, що для зберігання множинних наборів даних у ньому, крім множини процесорних елементів, має бути і множина модулів пам'яті. Крім того, у масиві повинна бути реалізована мережа взаємозв'язків, як між ПЕ, так і між процесорними елементами та модулями пам'яті. Таким чином, під терміном масив процесорних елементів розуміють блок, що складається з власне процесорних елементів, модулів пам'яті та мережі з'єднань.

Додаткову гнучкість при роботі з аналізованої системою забезпечує механізм маскування, що дозволяє залучати до операцій лише певну підмножина

ПЕ масиву. Маскування можливо як на стадії компіляції, і на етапі виконання, при цьому ПЕ, виключені шляхом встановлення в нуль відповідних бітів маски, під час виконання команди простоюють.

Масив процесорних елементів

У матричних SIMD-системах поширення набули два основних типу архітектурної організації масиву процесорних елементів.

У першому варіанті, відомому як архітектура типу «процесорний елемент-процесорний елемент» («ПЕ-ПЕ»), N процесорних елементів (ПЕ) пов'язані між собою комунікаційної мережею. Кожен ПЕ – це процесор з локальною пам'яттю. Процесорні елементи виконують команди, що отримуються з КМП по шині широкомовної розсилки та обробляють дані, як, що зберігаються в них локальною пам'яті так і вступники з КМП. Обмін даними між процесорними елементами виконується по комунікаційній мережі, на той час як шина введення / виведення служить для обміну інформацією між ПЕ та пристроями введення / виведення. Для трансляції результатів з окремих ПЕ в контролер масиву процесорних елементів служить шина результату. Завдяки використанню локальною пам'яті апаратні засоби ОС розглянутого типу можуть бути побудовані дуже ефективно. У багатьох алгоритмах дії з пересилання інформації з більшої частини локальні, тобто відбуваються між найближчими сусідами. З цієї причини архітектура, де кожен ПЕ пов'язаний тільки з сусідніми, дуже популярна.

Другий вид архітектури – «процесор -пам'ять». конфігурації двоспрямована мережа з'єднань пов'язує N процесорів з M модулями пам'яті. Процесори управляються КМП через широкомовну шину. Обмін даними між процесорами здійснюється як через мережу, так і через модулі пам'яті. Пересилання даних між модулями пам'яті та пристроями введення / виведення забезпечується шиною введення / виведення. Для передачі даних з конкретного модуля пам'яті КМП служить шина результату.

Векторні обчислювальні системи – системи класу SIMD, у яких та сама задана операція виконується відразу над усіма компонентами векторів (рис. 4.13).

У завданнях моделювання реальних процесів та об'єктів, для яких характерна обробка великих масивів чисел у формі з плаваючою коми, масиви видаються матрицями та векторами, а алгоритми їх обробки описуються в термінах матричних операцій. Як відомо, основні матричні операції зводяться до однотипних діям над парами елементів вихідних матриць, які, частіше всього, можна виробляти паралельно. В універсальних обчислювальні системи, орієнтовані на скалярні операції, обробка матриць виконується поелементно та послідовно. При великій розмірності масивів послідовна обробка елементів матриць займає занадто багато часу, що і призводить до неефективності

універсальних ОС для аналізованого класу завдань. Для обробки масивів потрібні обчислювальні засоби, що дозволяють за допомогою єдиною команди виробляти дія відразу над усіма елементами масивів – засоби векторної обробки.

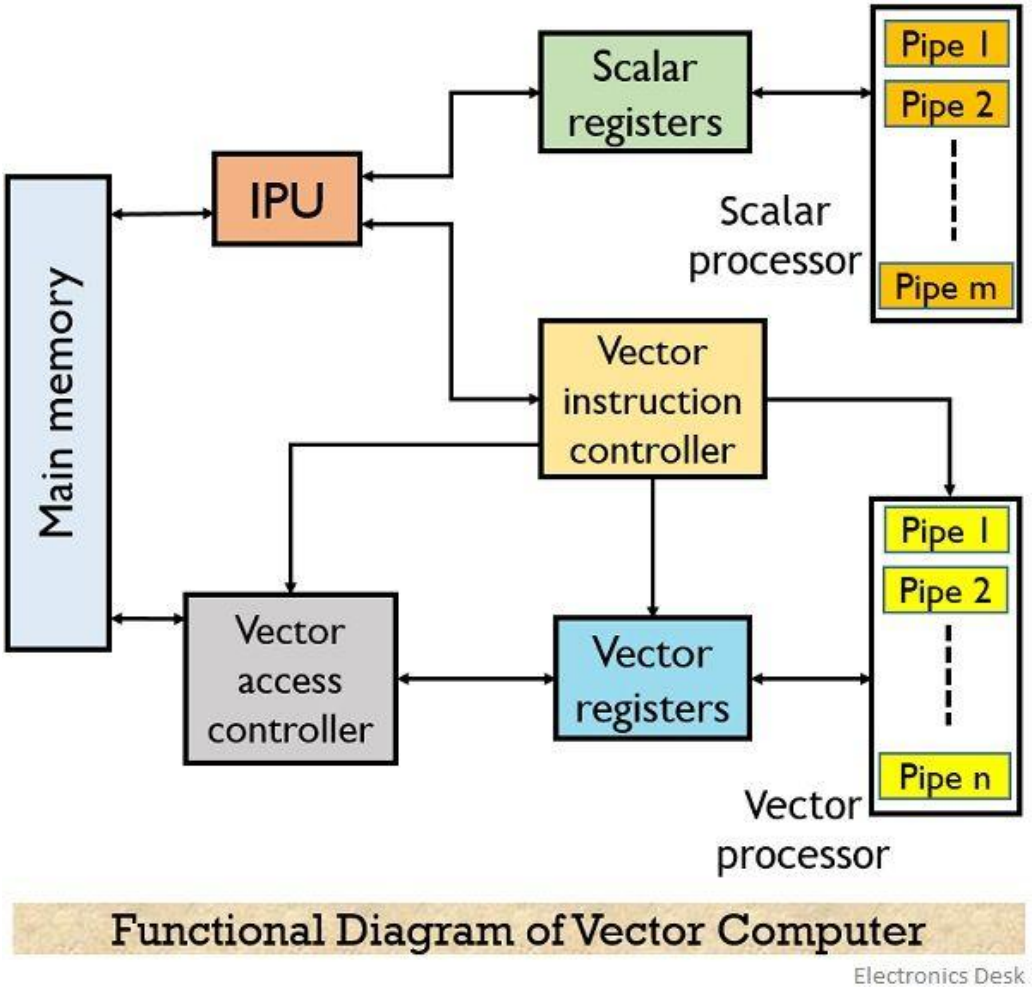

Рисунок 4.13 – Функціональна діаграма векторного комп'ютера [24]

У засобах векторної обробки під вектором розуміється одномірний масив даних (зазвичай у формі з плаваючою комою), розміщених у пам'яті ОС. Кількість елементів масиву називається довжиною вектора. Багатовимірні масиви вважаються наборами одновимірних масивів-векторів.

Дії над багатовимірними масивами враховують специфіку їх розміщення. Спосіб розміщення багатовимірного масиву впливає на крок зміни адреси елемента, що вибирається з пам'яті. Так, якщо матриця розташована у пам'яті построчно, адреси сусідніх елементів рядка різняться на одиницю, а для елементів стовпця крок дорівнює чотирьом. При розміщенні матриці по стовпцях одиниці буде дорівнює крок по стовпцю, а чотирьом -крок по рядку. У векторній концепції для позначення кроку, з яким елементи вектора витягуються з пам'яті, застосовують термін крок за індексом (stride).

Векторний процесор – це процесор, в якому операндами деяких команд можуть виступати масиви даних – вектори. вектор процесор може бути реалізований у двох варіантах. У першому він представляє собою додатковий блок до універсальної обчислювальної машині (системи). У другому – векторний процесор є основою самостійної ОС.

В архітектурі засобів векторної обробки використовується один з двох підходів – векторно-паралельний або векторно-конвеєрний.

У векторно-паралельному процесорі одночасні операції над елементами векторів проводяться за допомогою кількох функціональних блоків (ФБ) з плаваючою комою, кожен з яких відповідає за одну пару елементів.

У векторно-конвеєрному варіанті обробка елементів векторів виробляється одним конвеєрним ФБ. Операції з числами у формі з ПЗ достатньо складні, але піддаються розбиття на окремі кроки. Кожен етап обробки може бути реалізований за допомогою окремої ланки конвеєрного ФБ. Чергова пара елементів векторів-оперендів подається на вхід конвеєра як тільки звільняється його перша ланка.

Одночасні операції над елементами векторів можна проводити і за допомогою кількох конвеєрних ФБ. Такого роду обробка поєднує векторно-паралельний та векторно-конвеєрний підходи.

Архітектури векторної обробки «пам'ять-пам'ять» та «регістр-регістр» (рис.4.14).

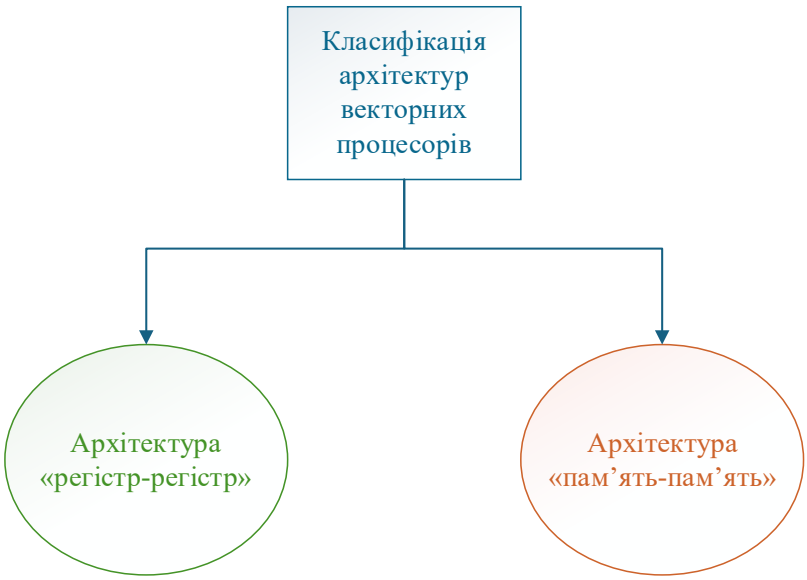

Рисунок 4.14 – Класифікація архітектур векторних процесорів

Важливим моментом в архітектурі векторних процесорів є спосіб доступу до операндів, оскільки вектори-операнди зберігаються у пам'яті ОС і туди ж міститься вектор-результат. Для відомих векторних ОС можна виділити два варіанти архітектури векторної обробки, відомі як «пам'ять-пам'ять» та «регістр-регістр».

У векторних процесорах з архітектурою «пам'ять-пам'ять» елементи векторів по черзі витягуються з пам'яті і відразу ж направляються у функціональний блок. У міру обробки елементи вектора результату, що з'являються на виході ФБ, відразу ж заносяться в пам’ять.

В архітектурі «регістр-регістр» операнди спочатку завантажуються з пам'яті у векторному регістри. вектор регістр представляє собою сукупність скалярних регістрів, об'єднаних у чергу типу FIFO, здатну зберігати 50-100 чисел із плаваючою комою (частіше всього – 64). Операція виконується над векторами, розміщеними у векторних регістрах операндів, а її результат спочатку заноситься у векторний регістр результату, а вже з нього переписується в пам’ять.

В обох структурах необхідно забезпечити необхідну послідовність вилучення елементів векторів-операндів з пам'яті та занесення елементів вектора- результату в пам’ять. Це завдання у векторному процесорі реалізується за допомогою генератора адрес, на виході якого формується адреса чергового елемент вектора в пам'яті. Спочатку на вхід генератора подається базова адреса – початкова адреса області пам'яті, що зберігає вектор елементів. Чергова адреса обчислюється шляхом збільшення попередньої адреси на величину кроку за індексом.

Для доступу до структурованих даних у пам'яті (масивам, векторам), у яких елементи з послідовно зростаючими індексами розміщуються в осередках з послідовно зростаючими адресами, пам'ять вигідніше будувати як блочну з розшаруванням. У цьому у разі адреси осередків чергуються за циклічною схемою (наступна адреса – в наступному банку пам'яті). Такий прийом дозволяє майже паралельно читати (записувати) елементи векторів в обох архітектури.

Перевага векторних процесорів «пам'ять-пам'ять» полягає у можливості обробки довгих векторів, в той час як у процесорах «регістр-регістр» доводиться розбивати довгі вектори на сегменти фіксованою довжини. На жаль, за гнучкість режиму «пам'ять-пам'ять» доводиться розплачуватися дещо великими витратами, відомими як час запуску, що являє собою тимчасовий інтервал між ініціалізацією команди і моментом, коли перший результат з'явиться на виході конвеєра.Великий час запуску в процесорах "пам'ять-пам'ять" обумовлено швидкістю доступу до пам'яті, яка набагато менше швидкості доступу до внутрішнього регістру.

Структура векторної обчислювальної системи.

В реальних задачах векторна обробка складає тільки частину загального обчислювального навантаження. Значну вагу мають і скалярні операції. За цією причини векторна ОС, окрім векторного процесора, містить ще й скалярний процесор. Як і належить для SIMD-системи, виконується єдина програма, що містить як скалярні, так і векторні команди. Програма та дані зберігаються у пам'яті ОС. Команди програми послідовно вибираються з пам'яті процесором

обробки команд, який спрямовує скалярні та векторні команди в скалярний або векторний процесор відповідно.

Для підвищення швидкості обробки векторів всі функціональні блоки векторні процесорів будуються по конвеєрній схемою, причому так, щоб кожен ступінь будь-якого з конвеєрів справлялася зі своєю операцією за один такт (кількість ступенів у різних функціональних блоках може бути різним). У деяких векторних ОС, цей підхід кілька вдосконалено – конвеєри у всіх функціональних блоках продубльовані.

Цікавою особливістю деяких ВП типу «регістр-регістр», є так зване зачеплення векторів (vector chaining або vector linking), коли векторний регістр результату однієї векторної операції використовується як вхідний регістр для наступної векторної операції. Така комбінація з послідовності множення та підсумовування характерна для операції згортки і зустрічається в багатьох векторних та матричних обчисленнях. Сутність зачеплення векторів у тому, що виконання векторної команди починається відразу, як тільки утворюються компоненти векторних операндів, що беруть у ній участь, не чекаючи завершення обчислення повного вектора операнда та занесення його у відповідний векторний регістр. Так утворюються ланцюжки операцій.

Паралельні векторні системи (PVP, Parallel Vector Processor) – MIMD-системи з однорідним доступом до розділяється пам'яті.Основною ознакою PVP-систем є наявність спеціальних векторно-конвеєрних процесорів, в яких передбачені команди однотипний обробки векторів незалежних даних, ефективно виконуються на конвеєрних функціональних пристрої

За своєю суті PVP- система – це симетрична мультипроцесорна система (SMP- система), де роль процесорних елементів виконують векторно-конвеєрні процесори.

PVP-система містить порівняно невелику кількість індивідуальних векторних процесорів, пов'язаних широкосмуговим комутатором. Завдяки таким комутаторам кількість ПЕ у системі може бути більше, ніж у SMP-системах з шинною організацією, і може досягати 512, хоча типові значення – 8-16 процесорних елементів.

PVP- системи орієнтовані на застосунки з таких галузей промисловості, як аерокосмічна, автомобільна, електронна, хімічна, енергетична тощо. Розробники PVP-систем пропонують користувачам компілятори з ефективними засобами автоматичною векторизації та розпаралелювання програмних обчислень. Отримуваний об'єктний код дозволяє найкраще використовувати потенціал багатьох векторних процесорних елементів. Крім того, потрібно враховувати, що передача даних у векторному форматі відбувається на два порядки швидше, ніж у скалярному. Як наслідок, витрати часу на взаємодію між паралельними потоками даних суттєво знижуються, що відчутно позначається на загальній продуктивності системи.

Системи з масовою паралельною обробкою.

Система з масовою паралельною обробкою (MPP, Massively Parallel Processing) – система, що складається з однорідних обчислювальних вузлів, що мають усі засоби для незалежного функціонування. У MPP-архітектурі реалізовано модель розподіленої пам'яті.

Основні причини появи систем з масовою паралельною обробкою – це, по - перше, необхідність побудови ОС з гігантською продуктивністю і, по- друге, прагнення розширити межі виробництва ОС у великому діапазоні продуктивності та вартості. Для MPP- системи, в якій кількість обчислювальних вузлів може змінюватися в широких межах, завжди реально підібрати конфігурацію із заздалегідь заданою обчислювальної потужністю та фінансовими вкладеннями.

MPP-система складається з множини однорідних обчислювальних вузлів, кількість яких може обчислюватися тисячами.Вузол містить повний комплекс пристроїв, необхідних для незалежного функціонування (процесор, пам'ять, підсистему введення / виведення, комунікаційне обладнання), тобто, по суті, є повноцінною обчислювальної машиною.Вузли об'єднані комунікаційної мережею з високою пропускний здатністю та малими затримками.

Роботу вузлів MPP- системи координує головна керуюча обчислювальна машина (хост -комп'ютер).Це може бути як окрема спеціалізована ОМ, так один з вузлів системи.В даному випадку функції головної ОМ покладаються на якийсь певний вузол ОС (протягом всього сеансу обчислень). В SMP-системі особливі повноваження одному з ПЕ надаються тільки на час завантаження системи, після чого він знову стає рівноправним елементом системи.

Якщо система містить виділений хост-комп'ютер, то повноцінна операційна система (ОС) функціонує тільки на ньому, а на вузли встановлюється її урізаний варіант, що підтримує лише функції ядра ОС. За відсутності головної ОМ повноцінна ОС встановлюється на кожен вузол MPP- системи. Так, кожен вузол функціонує під управлінням власної операційної системи.

Хост-комп'ютер (або його замінник з числа вузлів) розподіляє завдання між множиною підлеглих йому вузлів.Схема їх взаємодії досить проста:

- хост-комп'ютер формує чергу завдань, кожному з яких призначається деякий рівень пріоритету ;
- у міру звільнення вузлів їм передаються завдання з черги ;
- вузли сповіщають хост-комп'ютер про хід виконання завдання; зокрема про завершення виконання або про потребу у додаткових ресурсах;
- у хост-комп'ютера є засоби для контролю роботи підлеглих процесорів, у тому числі для виявлення позаштатних ситуацій, переривання виконання завдання у разі появи більше пріоритетною завдання і т.п.

У деякому наближенні має сенс вважати, що на головній ОМ виконується ядро операційної системи (планувальник завдань), але в підлеглих їй вузлах –

застосунки. Підпорядкованість може бути реалізована як на апаратному, так і на програмному рівні.

Процесори у вузлах мають доступ тільки до своєї локальною пам'яті. Для доступу до пам'яті іншого вузла пара вузлів (клієнт та «сервер») повинна обмінятися повідомленнями. Такий режим виключає можливість конфліктів, що виникають у поділяється пам'яті при одночасному зверненні до неї зі сторони кількох процесорів. Знімається також проблема когерентності кеш-пам'яті. Як слідство, з'являється можливість нарощування кількості процесорів до кількох тисяч. За цією причини основною ознакою, за якою обчислювальну систему відносять до MPP-типу, що часто служить кількість процесорів n. Чіткої межі не існує, але зазвичай при $n \geq 128$ вважається, що це вже MPP. Зрештою, відмінною рисою MPP є її архітектура, що дозволяє досягти найвищою продуктивності. Завдяки властивості масштабованості MPP- системи є сьогодні лідерами за досягнутою продуктивності.

Тим не менш, розпаралелювання обчислень у MPP-системах є важкий завданням. Достатньо складно знайти завдання, які зуміли б ефективно завантажити велику кількість обчислювальних вузлів. Сьогодні не так вже багато застосунків можуть ефективно виконуватися на MPP- системі. З'являється також проблема переносимості програм між системами з різною архітектурою. Ефективність розпаралелювання у багатьох випадках сильно залежить від деталей архітектури MPP- системи, наприклад топології з'єднання обчислювальних вузлів.

Самою ефективною була б топологія, в якій будь-який вузол міг б безпосередньо зв'язатися з будь-яким іншим вузлом, але в ОС на основі MPP це технічно неможливо реалізувати. Якщо в перших MPP- комп'ютерах використовувалися топології двовимірної решітки та гіперкуба, то в сучасних найбільш масштабованих і продуктивних ОС, наприклад сімействах, застосовують тривимірний тор. Діаметр такої мережі для різних моделей згаданих систем лежить у діапазоні від 20 до 60, він суттєво впливає на затримки передачі повідомлень. Коли множинні повідомлення починають розділяти ресурси мережі, даний ефект (стосовно до повідомлень великого розміру) стає очевидним. Таким чином, при розподілі завдань по вузлам ПС необхідно враховувати топологію системи.

Час передачі інформації від вузла до вузла залежить від стартової затримки та швидкості передачі. У будь -якому випадку, за час передачі процесорні вузли встигають виконати багато команд, і таке співвідношення (швидкості процесорних вузлів та передавальної системи), ймовірно, буде зберігатися – прогрес у продуктивності процесорів набагато більший, ніж у пропускній здібності каналів зв'язку. Тому інфраструктура каналів зв'язку в MPP-системах є об'єктом найбільш пильного уваги розробників.

Слабким місцем MPP є хост-комп'ютер – при виході його з ладу вся система виявляється непрацездатною. Підвищення надійності головною ОМ лежить на шляхах спрощення апаратури хост-комп'ютера та/ або її дублювання.

Незважаючи на всі складнощі, сфера застосування ОС з масовим паралелізмом постійно розширюється. Різні системи цього класу експлуатуються у багатьох провідних суперкомп'ютерні центри світу.

Головні особливості, за якими обчислювальну систему відносять до класу MPP, можна сформулювати наступним чином:

- обчислювальний вузол має усі засоби для незалежного функціонування (стандартні мікропроцесори з кеш-пам'яттю, локальна пам'ять, підсистема введення / виведення);
- кожен вузол містить мережевий адаптер, який використовується для об'єднання вузлів;
- реалізовано модель розподіленої пам'яті – доступ до пам'яті інших вузлів забезпечується шляхом асинхронного обміну повідомленнями;
- мережа з'єднань зазвичай проектується під конкретну систему для забезпечення високою пропускної здатності та малих затримок;
- система добре масштабується (до тисяч вузлів);
- робота системи координується головною ОМ (хост-комп'ютером) або одним з вузлів, що виконують роль головної ОМ;
- на кожному вузлі встановлена операційна система або її урізаний варіант, якщо є хост-комп'ютер з повноцінною операційної системою ;
- обчислення представляють собою множину процесів, що мають окремі адресні простори.

Кластерні обчислювальні системи

Кластер – група взаємно з'єднаних обчислювальних систем (вузлів), що працюють спільно як складові єдиного обчислювального ресурсу, створюючи ілюзію наявності єдиною ОМ (рис.4.15). Для зв'язку вузлів використовується одна з стандартних мережевих технологій (Fast / Gigabit Ethernet, Myrinet) на базі шинний архітектури або комутатора. Напочатку перед кластерами ставилися дві завдання: досягти великий обчислювальної потужності та забезпечити підвищену надійність ОС, а на сьогодні кластеризація – це один з найсучасніших напрямків в області створення обчислювальних систем. Крім терміну «кластерні обчислення», досить часто застосовують такі назви: кластер робочих станцій (workstation cluster), гіперобчислення (hypercomputing), паралельні обчислення на базі мережі (network-based concurrent computing), ультраобчислення (ultracomputing).

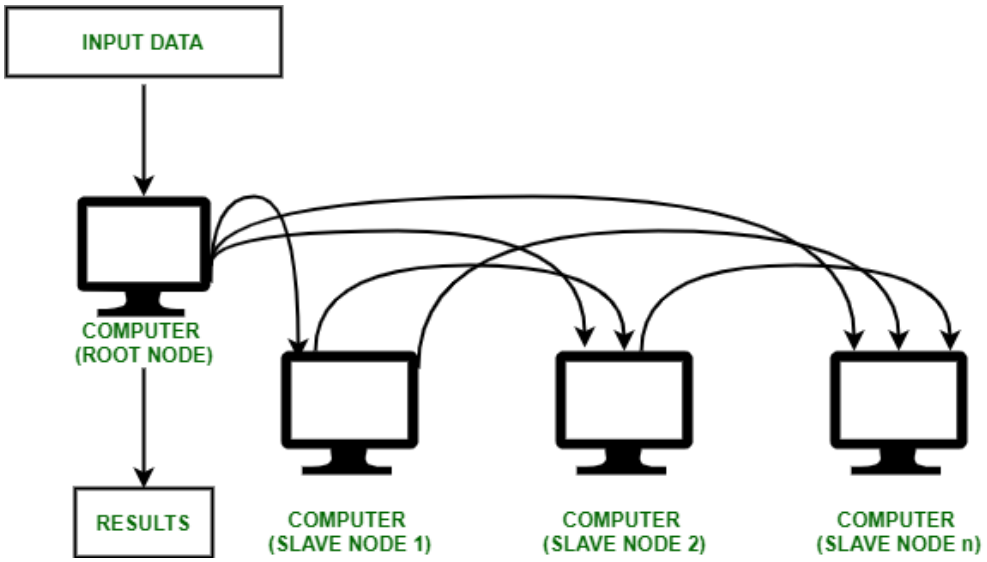

Рисунок 4.15 – Схема кластерних обчислень [25]

Архітектура кластерної система багато в чому схожа на архітектуру MPP-систем.

Той самий принцип розподіленої пам'яті, використання як обчислювальних вузлів закінчених обчислювальних машин, великий потенціал для масштабування системи та цілий ряд інших особливостей. У першому наближенні кластерну технологію можна розглядати як розвиток ідей масових паралельних обчислень. З іншої сторони, багато рис кластерної архітектури дають підстава рахувати її самостійним напрямом у галузі MIMD-систем.

В якості вузла кластера може виступати як однопроцесорна ОМ, так і ОС типу SMP (логічна SMP-система, що представлена як єдина ОМ). Як правило, це не спеціалізовані пристрої, пристосовані під використання у обчислювальній системі, як у MPP, а серійні обчислювальні машини або системи. Ще одна особливість кластерної архітектури полягає в тому, що в єдину систему об'єднуються вузли різного типу, від персональних комп'ютерів до потужних ОС. Кластерні системи з однаковими вузлами називають гомогенними кластерами, а з різнотипними вузлами – гетерогенними кластерами.

Використання машин масового виробництва суттєво знижує вартість ОС, а можливість варіювання різних за типом вузлів дозволяє отримати необхідну продуктивність за прийнятну ціну. Важливо й те, що вузли можуть функціонувати самостійно та окремо від кластера. Для цього кожен вузол працює під управлінням своєї операційної системи. Частіше всього використовуються стандартні ОС: Linux, FreeBSD, Solaris та версії Windows, що продовжують напрям Windows NT.

Вузли у кластерній системі об'єднані високошвидкісний мережею. Рішення можуть бути простими, заснованими на апаратурі Ethernet, або складними з високошвидкісними мережами пропускний здібності в сотні мегабайтів в секунду (Мбіт /с). В принципі, за основу кластерної системи може бути взята

стандартна локальна мережа (або мережа більшого масштабу), із збереженням прийнятих у ній протоколів (правил взаємодії). Апаратурні зміни можуть не знадобитися або, у гіршому випадку, зводяться до заміни комунікаційного обладнання на більш продуктивне. При з'єднанні машин у кластер майже завжди підтримуються прямі міжмашинні зв'язки.

Обчислювальні машини (системи) в кластері взаємодіють відповідно до одного з двох транспортних протоколів. Перший їх, протокол TCP (Transmission Control Protocol), оперує потоками байтів, гарантуючи надійність доставки повідомлення. Другий – UDP (User Datagram Protocol) намагається посилати пакети даних без гарантії їх доставки. В останнє час застосовують спеціальні протоколи, які працюють набагато краще, наприклад Virtual Interface Architecture (VIA).

При обміні інформацією використовуються два програмні методи: передачі повідомлень та розподіленої, спільно використовуваної пам'яті. Перший спирається на явну передачу інформаційних повідомлень між вузлами кластеру. В альтернативному варіанті також відбувається пересилання повідомлень, але рух даних між вузлами кластера приховано від програміста.

Підключення вузлів до мережі здійснюється за допомогою мережевих адаптерів. Враховуючи роль комунікацій для зв'язку ядра обчислювального вузла з мережею використовується виділена шина введення / виведення. До цієї шині також підключаються локальні диски. Наявність таких дисків характерно для кластерних систем, але не характерно для MPP-систем. Елементи обчислювального ядра об'єднуються за допомогою локальної системної шини. Зв'язок між цією шиною та шиною введення / виведення забезпечує міст.

Також невід'ємною частиною кластера є спеціалізоване програмне забезпечення (ПЗ), що організує безперебійну роботу при відмові одного або кількох вузлів. Таке ПЗ повинно бути встановлено на кожен вузол кластеру. Воно реалізує механізм передачі повідомлень над стандартними мережевими протоколами і може розглядатися як частина операційної системи. Саме завдяки спеціалізованому ПЗ група ОМ, об'єднаних мережею, перетворюється на кластерну обчислювальну систему. Кластерне ПЗ перерозподіляє обчислювальне навантаження при відмові одного або кількох вузлів кластера, а також відновлює обчислення при збої у вузлі. ПЗ кожного вузла постійно контролює працездатність всіх інших вузлів. Цей контроль заснований на періодичній розсилці кожним вузлом сигналу, відомого як keepalive («поки живий») або heartbeat («серцебиття»). Якщо сигнал від деякого вузла не надходить, то вузол вважається таким, що вийшов з ладу ; йому не дається можливість виконувати введення / виведення, його диски та інші ресурси (включаючи мережеві адреси) перепризначаються іншим вузлам, а виконувані їм програми перезапускаються в інших вузлах. За наявності у кластері спільно використовуваних дисків, кластерне ПЗ підтримує єдину файлову систему.

Кластери забезпечують високий рівень доступності – у них відсутні єдина операційна система та спільно використовувана пам'ять, тобто немає проблеми когерентності кешів.

При створенні кластерних систем використовується один з двох підходів.

Перший підхід застосовується для побудови невеликих кластерних систем, наприклад, на базі невеликих локальних мереж організацій або їх підрозділів. У такому кластері кожна ОМ продовжує працювати як самостійна одиниця, одночасно виконуючи функції вузла кластерною системи.

Другий підхід орієнтований на використання кластерної системи у ролі потужного обчислювального ресурсу. В якості вузлів кластера використовуються лише системні блоки обчислювальних машин, компактно розміщені у спеціальних стійках. Управління системою та запуск завдань здійснює повнофункціональний хост – комп'ютер, що підтримує дискову підсистему кластера та різноманітне периферійне обладнання. Відсутність у вузлів власної периферії суттєво здешевлює систему.

Кластери добре масштабуються шляхом додавання вузлів, що дозволяє досягти найвищих показників продуктивності. Завдяки цією особливості архітектури кластери з сотнями та тисячами вузлів позитивно зарекомендували себе на практиці.

Чотири переваги, що досягаються за допомогою кластеризації:

- Абсолютна масштабованість. Можливо створення великих кластерів, що перевершують за обчислювальною потужності навіть самі продуктивні поодинокі ОМ. Кластер у змозі містити десятки вузлів, кожен з яких представляє собою мультипроцесор.
- Нарощувана масштабованість. Кластер будується так, що його можна нарощувати, додаючи нові вузли невеликими порціями. Таким чином, користувач може почати з невеликої системи, розширюючи її за необхідності.
- Високий коефіцієнт готовності. Оскільки кожен вузол кластера – самостійна ОМ або ОС, відмова одного з вузлів не призводить до втрати працездатності кластера. У багатьох системах відмовостійкість автоматично підтримується програмним забезпеченням.
- Чудове співвідношення ціна / продуктивність. Кластер будь-якої продуктивності можна створити, з'єднуючи стандартні «будівельні блоки», при цьому його вартість буде нижче, ніж у одиночної ОМ з еквівалентною обчислювальною потужністю.

Водночас потрібно згадати і основний недолік, властивий кластерним системам, – взаємодія між вузлами кластера займає набагато більше часу, ніж у інших типах ОС.

Кластери великих SMP-систем.

Величезний потенціал масштабування, властивий кластерній архітектурі, робить її дуже перспективним напрямом в області створення високопродуктивних обчислювальних систем. Масштабування можливо як за

рахунок збільшення числа вузлів, так і шляхом застосування як вузлів не одиночних ОМ, а також добре масштабованих обчислювальних систем, зазвичай SMP – типу. Цей напрямок отримав настільки широкий розвиток, що було виділено в окрему Групу MIMD – пристроїв – Constellations. Термін, що визназначений назвою однієї з перших ОС даного типу – Sun "Constellation", – можна перекласти як "сузір'я ".

Constellation-система – це кластер, вузлами якого служать SMP -системи. В якості відмінної ознаки виступає кількість процесорних елементів у вузлі. Спочатку приймалося, що система відноситься до Constellation -систем, якщо число вузлів у кластері з SMP-систем менше або дорівнює кількості процесорних елементів у SMP- системі вузла. Для сучасних систем з великим кількістю вузлів така умова не завжди дотримується, тому в даний час час умовою зарахування ОС до Constellation -систем служить число ПЕ у вузлі – воно повинно бути більше (або дорівнювати) 16. Системи, що не відповідають даному умові, вважаються класичними кластерними системами.

Перспективність Constellation-систем обумовлена вдалим поєднанням переваг розподіленості пам'яті (можливості нарощування кількості вузлів) та розподілу пам'яті (ефективного доступу множини ПЕ до пам'яті).

Формально структура Constellation-системи повністю відповідає кластерній архітектурі, однак явна спрямованість на високу продуктивність може відображатись в деяких конструктивних рішеннях.

CISC (Complete Instruction Set Computing) – тип архітектури процесор з повним набором команд. Основоположником CISC-архітектури вважається фірма IBM із архітектурою IBM/360. При цьому підході виконання будь-який як завгодно складною команди з системи команд процесора реалізується апаратно всередині самого процесора.

Основну ідею CISC-архітектури відображає її назва – "повний набір команд". У цій архітектурі прагнуть мати окрему машинну команду для кожного можливої (типової) дії з обробки даних.

Історично CISC-архітектура була однієї з перших. Вдосконалення процесорів йшло шляхом створення ОМ, здатних виконувати як можна більше різних команд. Це спрощувалося роботу програмістів, які писали програми мовою асемблера (тобто практично на рівні машинних команд). Використання складних команд дозволяло скоротити розмір та час розробки програми.

У результаті склалися наступні риси організації CISC- процесорів:

- велика кількість різних машинних команд (сотні), кожна з яких виконується за кілька тактів центрального процесора ;
- пристрій управління з програмованою логікою ;
- невелика кількість регістрів спільного призначення ;
- різні формати команд з різною довжиною ;

- переважання двоадресної адресації ;
- розвинений механізм адресації операндів, що включає різні методи непрямої адресації.

CISC- підхід, однак, призвів до того, що деякі команди стало неможливо виконувати чисто апаратними засобами (при розумній складності таких засобів). В результаті в процесорах з'явилися блоки, які «на льоту» замінюють найбільш складні команди послідовностями з більш простіших команд. Мало того, практика показала, що багато складних команд при написанні програм виявлялися просто незатребуваними. І з рештою, через високу складність команд та їх розмаїття пристрій управління ОМ доводилося будувати тільки на основі програмованої логіки, тобто із застосуванням «повільної» керуючої пам'яті. Ця обставина суттєво обмежувала можливості нарощування тактової частоти процесора. Усі ці фактори призвели до повороту у бік RISC-архітектури. Водночас цілий ряд безперечних переваг CISC-архітектури зберігають її актуальність (перш всього, в очах розробників програмних додатків). Саме тому ведучі фірми-виробники ОМ (Intel, AMD, IBM та ін.) у своїх останніх розробках, як і раніше, не відмовляються від CISC- підходу.

Сучасним прикладом CISC є архітектура x86. Процесори сімейства Intel 8086, 80286, i386, Pentium і аж до сучасних Core i9 – мають архітектуру CISC. Характерними для цієї архітектури є: десятки регістрів, сотні інструкцій, складні схеми декодування, підтримка інструкцій із пам’яттю, умовами та адресацією.

Еволюція архітектури x86/IA-32 – x86-64

1. IA-32 (x86, 32-бітна архітектура)

- Розроблена Intel у 1985 році (процесор 80386).
- Підтримує максимум 4 ГБ оперативної пам’яті.
- Використовує сегментну та сторінкову адресацію.
- Має обмежену кількість регістрів, що ускладнює оптимізацію коду.

2. Перехід до x86-64 (AMD64, 64-бітна архітектура)

- Розроблена AMD у 2003 році (процесор Athlon 64).
- Підтримує до 256 ТБ пам’яті та збільшену кількість регістрів.
- Висока продуктивність завдяки ефективнішому управлінню пам’яттю.
- Зворотна сумісність із 32-бітними програмами.

NB! Починаючи з Pentium Pro (1995), всередині CISC-процесори трансформують інструкції в RISC-подібні мікрооперації – тобто використовують RISC-ядерця для обробки.

Хоча сьогодні x86-64 є домінуючою архітектурою в персональних комп’ютерах та серверних системах, вона поступово втрачає позиції у мобільному сегменті через зростання популярності енергоефективних рішень ARM.

RISC (Restricted (Reduced) Instruction Set Computer – комп'ютер із скороченим набором команд) – архітектура процесора, в якій швидкодія збільшується за рахунок спрощення інструкцій, щоб їх декодування було більше простим, а час виконання – коротше. У процесорах з RISC-архітектурою використовується обмежений набір швидких команд. Кожна команда RISC-процесора повинна виконуватися за один машинний такт. Це полегшує підвищення тактовий частоти та робить більш ефективною суперскалярність (розпаралелювання інструкцій між декількома виконавчими блоками) У таких мікропроцесорах. міститься менше кількість транзисторів, що знижує їх вартість і енергоспоживання.При цьому, як правило, зростає їх продуктивність. Архітектура RISC є основою сучасних високопродуктивних ОМ.

Ідея RISC-архітектури була висунута в 1975 році Джоном Коком з IBM Research і вперше реалізована у 1980 році. Суть концепції RISC полягає у зведенні набору команд ОМ до найбільш вживаних найпростіших команд. Це дозволяє спростити схемотехніку процесора і досягти різкого скорочення часу виконання кожною з "простих" команд. Більше складні команди реалізуються як підпрограми, складені з швидких «простих» команд.

Архітектура RISC- процесорів характеризується наявністю команд фіксованою довжини великої кількості регістрів, операцій типу регістр-регістр, а також відсутністю непрямої адресації.

Головні зусилля в архітектурі RISC спрямовані на побудову максимально ефективного конвеєра команд, тобто такого, де всі команди витягуються з пам'яті та надходять у центральний процесор (ЦП) на обробку у вигляді рівномірного потоку, причому жодна команда не повинна перебувати у стані очікування, а ЦП повинен залишатися завантаженим протягом всього часу. Крім того, ідеальним буде варіант, коли будь-який етап циклу команди виконується протягом одного тактового періоду.

Для технології RISC характерна порівняно проста структура пристрою управління. Площа, що виділяється на кристалі мікросхеми для його реалізації, значно менше. Як наслідок, з'являється можливість розмістити на кристалі велику кількість регістрів ЦП. Крім того, залишається більше місця для інших вузлів ЦП та для додаткових пристроїв: кеш-пам'яті, блоку арифметики з плаваючою комою, частини основний пам'яті, блоку управління пам'яттю, портів введення / виведення.

Уніфікація набору команд, орієнтація на конвеєрну обробку, уніфікація розміру команд та тривалості їх виконання, усунення періодів очікування в конвеєрі – всі ці фактори позитивно позначаються на загальному швидкодії.

Недоліки RISC прямо пов'язані з деякими перевагами цією архітектури. Принциповий недолік – скорочена кількість команд: виконання ряду функцій доводиться витрачати кілька команд замість однієї в CISC. Це подовжує код програми, збільшує завантаження пам'яті та трафік команд між пам'яттю і ЦП.

Дослідження показали, що RISC- програма в середньому на 30% довша CISC-програми, що реалізує ті ж функції.

Хоча велика кількість регістрів дає суттєві переваги, саме по собі це ускладнює схему декодування номера регістру, тим самим збільшуючи час доступу до регістрів.

Пристрій управління з апаратною логікою, що реалізовано в більшості RISC-систем, менш гнучкий і більш схильний до помилок, що ускладнює пошук та виправлення помилок, особливо при виконанні складних команд.

Насьогодні базована на RISC архітектура ARM є основою мобільних пристроїв.

ARM (Advanced RISC Machine) – це 32/64-бітна RISC-архітектура, розроблена компанією ARM Ltd., що використовується у мобільних пристроях, вбудованих системах, енергоефективних серверних рішеннях, та ліцензується багатьма компаніями (Apple, Qualcomm, Samsung, MediaTek).

Розглянемо декілька критеріїв, що відрізняють в архітектури CISC та RISC

Критерій	CISC	RISC
Кількість інструкцій	Багато	Менше
Складність інструкцій	Висока	Низька
Робота з пам’яттю	Пряма в інструкціях	Через регістри
Виконання/конвеєризація	Багато тактів / складна	Зазвичай один такт/ проста
Енергоефективність	Низька	Вища
Розмір програми	Менший	Більший
Приклад	Intel x86	ARM, MIPS, RISC-V

VLIW (Very Long Instruction Word – дуже довга машинна команда) – архітектура процесорів, що характеризується можливістю об'єднання кількох простих команд у так звану зв'язку. Команди, що входять до її складу, повинні бути незалежні один від одного і виконуватись паралельно. Так, з кількох незалежних машинних команд транслятор формує одне «дуже довге командне слово».

Архітектура з командними словами надвеликий довжини або зі наддовгими командами відома з початку 80- х ряду університетських проектів. Ідея VLIW базується на тому, що завдання ефективного планування паралельного виконання команд покладається на "розумний" компілятор. Такий компілятор спочатку аналізує вихідну програму. Мета аналізу: виявити всі команди, які можуть бути виконані одночасно, причому так, щоб між командами не виникали

конфлікти. У ході аналізу компілятор може навіть частково імітувати виконання аналізованої програми. На наступному етапі компілятор намагається об'єднати такі команди в пакети (зв'язки), кожен з яких розглядається як одна наддовга команда. Об'єднання кількох простих команд в одну наддовгу провадиться за такими правилами:

- кількість простих команд, що об'єднуються в одну команду надвеликою довжини, дорівнює кількості наявних у процесорі функціональних (виконавчих) блоків (ФБ);
- у наддовгу команду входять тільки такі прості команди, які виконуються різними ФБ, тобто забезпечується одночасне виконання всіх складових наддовгою команди.

Довжина наддовгої команди зазвичай становить від 256 до 1024 бітів. Така метакоманда містить кілька полів (за кількістю утворюючих її простих команд), кожне з яких описує операцію для конкретного функціонального блоку.

В якості простих команд, що утворюють наддовгу, зазвичай використовуються команди RISC – типу. Максимальна кількість полів у наддовгій команді дорівнює числу обчислювальних пристроїв і зазвичай коливається в діапазоні від 3 до 20. Усі обчислювальні пристрої мають доступ до даних, що зберігаються в єдиному багатопортовому регістровому файлі.

VLIW-архітектуру можна розглядати як статичну суперскалярну архітектуру. Мається на увазі, що розпаралелювання коду проводиться на етапі компіляції, а не динамічно під час виконання. Те, що у виконуваній наддовгій команді виключено можливість конфліктів, дозволяє гранично спростити апаратуру VLIW- процесора і, як наслідок домогтися більш високого швидкодії. Переважна більшість цифрових сигнальних процесорів та мультимедійних процесорів з продуктивністю більше 1 млрд операцій /с базується на VLIW-архітектурі.

Двома проблемами VLIW-архітектури є:

1) ускладнення регістрового файлу і, перш за все, зв'язків цього файлу з обчислювальними пристроями ;

2) проблеми створення компіляторів, здатних знайти у програмі незалежні команди, зв'язати такі команди в довгі рядки та забезпечити їх паралельне виконання.

EPIC (Explicitly Parallel Instruction Computing) – архітектура процесора з явним паралелізмом команд. Термін введений у 1997 році фірмами HP та Intel для розроблюваної архітектури Intel Itanium. EPIC дозволяє мікропроцесору виконувати інструкції паралельно, спираючись на роботу компілятора, а не виявляючи можливість паралельною роботи інструкцій за допомогою спеціальних схем. Це дозволяє спростити масштабування обчислювальної потужності процесора без збільшення тактовий частоти.

В архітектурі EPIC, що виникла як спільна розробка фірм Intel та Hewlett-Packard у 1997 році, реалізований новий підхід, що є удосконаленим варіантом технології VLIW. Першим представником даної стратегії став мікропроцесор Itanium компанії Intel. Корпорація Hewlett-Packard також реалізує даний підхід у своїх розробках.

В архітектурі EPIC, яка у виробах Intel отримала назву IA-64 (Intel Architecture – 64), передбачається наявність у процесорі ста двадцяти восьми 64-розрядних регістрів спільного призначення та сто двадцяти восьми 80-розрядних регістрів з плаваючою комою. Крім того, процесор IA-64 містить 64 однобітових. регістра предикатів. Команди упаковуються (групуються) компілятором у наддовгу команду- зв'язку (bundle) довжиною в 128 розрядів. Логіка видачі команд на виконання складніше, ніж у традиційних процесорах типу VLIW, але набагато простіше, ніж у суперскалярних процесорів з невпорядкованою видачею.

Особливостями архітектури EPIC є:

- велика кількість регістрів ;
- масштабованість архітектури до великого кількості функціональних блоків. Цю властивість представники компаній Intel та HP називають спадково масштабованою системою команд (inherently scaleable instruction set);
- явний паралелізм у машинному коді. Пошук залежностей між командами здійснює не процесор, а компілятор ;
- предикація – команди з різних гілок умовного пропозиції забезпечуються полями предикатів (полями умов) та запускаються паралельно ;
- попереднє завантаження – дані з повільною основний пам'яті завантажуються заздалегідь.

На думку фахівців Intel і HP, концепція EPIC, зберігаючи всі переваги архітектурної організації VLIW, вільна від більшості її недоліків.

Окремо слід згадати технологію Hyper-threading (Hyper-Threading Technology, HTT) – це розробка компанії Intel, що вперше з'явилася в процесорах Intel Xeon та пізніше додана в процесори Pentium 4. Ця технологія підвищує ефективність процесора, дозволяючи виконувати два потоки інструкцій паралельно.

В основі гіперпотоковий технології, розробленої фірмою Intel та реалізованої в мікропроцесорі Pentium 4, лежить те, що сучасні процесори в більшості своєму є суперскалярними та багатоконвеєрними, тобто. виконання команд у них йде паралельно, за етапами та на кількох конвеєрах відразу. Гіперпотокова обробка покликана розкрити цей потенціал таким чином, щоб функціональні блоки процесора були б максимально завантажені. Ця мета досягається за рахунок поєднання відповідних апаратних та програмних засобів.

Виконувана програма розбивається на два паралельні потоку (threads). Завдання компілятора (на стадії підготовки програми) та операційної системи (на етапі виконання програми) полягає у формуванні таких послідовностей незалежних команд, які процесор міг б обробляти паралельно, по можливості заповнюючи функціональні блоки, не зайняті одним із потоків, відповідними командами з іншого, незалежного потоку. Так операційна система, що підтримує гіперпотокову технологію, сприймає фізичний суперскалярний процесор як два логічні процесора та організує надходження на ці два процесори двох незалежних потоків команд. Відповідно і процесор із підтримкою технології Hyper-Threading емулює роботу двох однакових логічних процесорів, приймаючи команди, спрямовані для кожного з них. Це не означає, що в процесорі. є два обчислювальні ядра – обидва логічних процесора конкурують за ресурси єдиного обчислювального ядра. Наслідком цієї конкуренції є більш ефективне завантаження всіх ресурсів процесора.

У процесі обчислень фізичний процесор розглядає обидва потоки команд та по черзі запускає на виконання команди то з одного, то з іншого, чи відразу їх двох, якщо є вільні обчислювальні ресурси. Жоден з потоків не вважається пріоритетним. При зупинці одного з потоків (в очікуванні будь- якої події або в результаті зациклювання) процесор повністю перемикається на другий потік. Можливість чергування команд з різних потоків складає принципову відмінність між гіперпотоковою та макропотоковою обробкою.

Наявність лише одного обчислювального ядра не дозволяє досягти подвоєної продуктивності, проте за рахунок більшої віддачі від усіх внутрішніх ресурсів загальна швидкість обчислень суттєво зростає. Це особливо відчувається, коли потоки містять команди різних типів, тоді уповільнення обробки в одному з них компенсується великим обсягом робіт, виконаних в іншому потоці.

Слід враховувати, що ефективність технології Hyper-Threading залежить від роботи операційної системи, оскільки поділ команд на потоки здійснює саме вона.

Організація ієрархії пам’яті: кеш, ОЗП, дискова пам’ять.

Сучасні комп’ютерні системи потребують ефективного управління пам’яттю, оскільки швидкість обробки даних безпосередньо залежить від того, наскільки швидко процесор отримує необхідну інформацію. Через значну різницю в швидкості між процесором та основною пам’яттю була розроблена ієрархічна структура пам’яті, яка дозволяє оптимізувати процес обробки даних. Так, ієрархія пам’яті будується за принципом компромісу між швидкістю доступу, обсягом та енергоспоживанням. Чим ближче пам’ять розташована до процесора, тим вона швидша, але її обсяг менший, а вартість реалізації значно вища.

Ієрархія пам’яті передбачає кілька рівнів зберігання даних, кожен з яких має свою швидкість, обсяг та вартість реалізації. Найближче до процесора розташована кеш-пам’ять, яка забезпечує надшвидкий доступ до часто використовуваних даних. Далі йде оперативна пам’ять (ОЗП, RAM), яка слугує основним сховищем для активних програм та операційної системи. Для довготривалого зберігання даних використовуються жорсткі диски (HDD) та твердотільні накопичувачі (SSD, NVMe SSD), які хоч і значно повільніші за оперативну пам’ять, але дозволяють зберігати великі обсяги інформації. Отже, основними видами пам’яті є кеш-пам’ять, оперативна, постійна та зовнішня, а її організація базується на ієрархічній структурі (рис. 4.16):

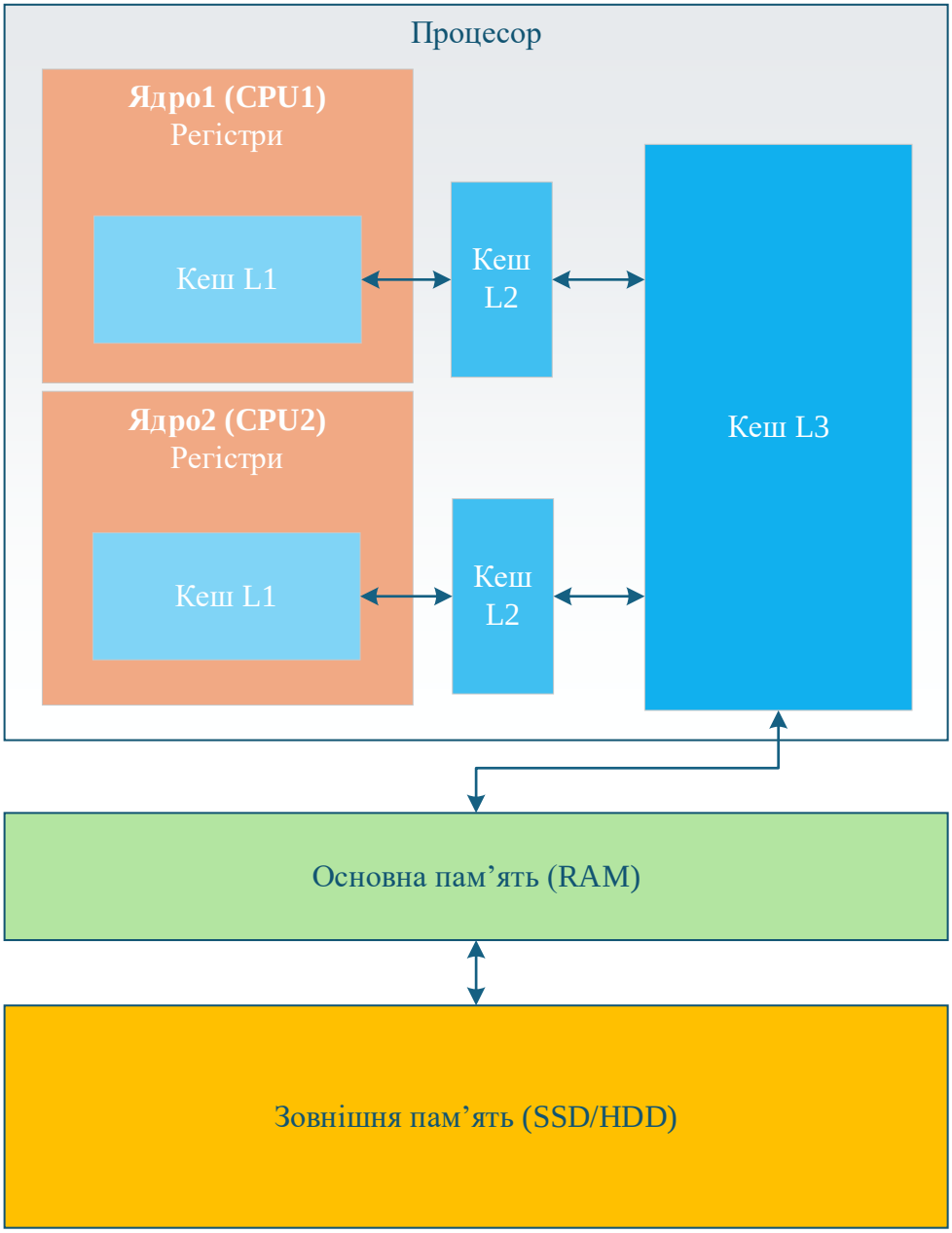

Рисунок 4.16 – Ієрархія пам’яті

- **Регістрова пам'ять** (CPU Registers) – це тип пам'яті, в якій зберігаються та приймаються дані, які негайно зберігаються в процесорі. Найчастіше використовуються регістри: акумулятор, лічильник програм, адресний регістр тощо.
- **Кеш-пам’ять** (cache memory L1, L2, L3) – це найшвидша пам'ять, яка розташовується найближче до процесора (зазвичай на тому ж кристалі) де дані тимчасово зберігаються для забезпечення найвищої швидкості доступу. Сучасні процесори мають 3 рівні кеш-пам’яті:
  - **L1 Cache** – найшвидший і найменший, розташований безпосередньо в ядрі процесора. Зберігає найчастіше використовувані дані та інструкції.
  - **L2 Cache** – більший за L1, може бути як у тому ж ядрі, так і окремим блоком на кристалі.
  - **L3 Cache** – спільний для кількох ядер, розташований на кристалі процесора.
- **Основна або оперативна пам’ять** (англ. random-access memory, RAM) – це пам'ять, яка використовується процесором для зберігання даних і програм під час їх виконання. Вона розташовується на материнській платі у вигляді окремих модулів, має невеликий об’єм (набагато більший ніж кеш , але і набагато менший ніж зовнішня пам’ять), і після вимкнення живлення дані втрачаються.
- **Дискова пам’ять** (SSD/HDD) – це зовнішня пам’ять комп’ютера, яка служить для довготривалого зберігання даних, таких як операційна система, програми, файли користувача, навіть за відсутності живлення. Доступ до цієї пам’яті повільніший, але зазвичай вона забезпечує великий обсяг зберігання.
- **Постійна пам’ять** (ROM, Read-Only Memory) – використовується для зберігання базових програм (наприклад, BIOS або UEFI), дані в цій пам’яті залишаються незмінними після вимкнення живлення, оскільки зазвичай мають власне резервне джерело живлення.

Ієрархія пам’яті дозволяє досягти оптимального балансу між вартістю компонентів, швидкістю доступу до даних і їхнім обсягом.

Кеш-пам’ять

Кеш-пам’ять є найшвидшим рівнем пам’яті в комп’ютерній системі та використовується для зберігання часто використовуваних даних та команд. Вона розташована безпосередньо в процесорі або поруч із ним, що дозволяє значно скоротити час очікування інформації.

Кеш-пам’ять працює за принципом локальності доступу, що означає, що якщо процесор нещодавно використовував певні дані або команди, то ймовірність того, що вони знадобляться знову, є високою. Завдяки цьому дані, які вже були завантажені в кеш, можуть бути використані повторно без необхідності звернення до повільнішої оперативної пам’яті.

Сучасні процесори використовують багаторівневу кеш-пам’ять, яка поділяється на кілька рівнів:

- L1-кеш – найшвидший, але найменший за обсягом рівень кеш-пам’яті, розташований безпосередньо в кожному ядрі процесора.

- L2-кеш – більшого розміру, але повільніший, ніж L1. Може бути як індивідуальним для кожного ядра, так і спільним для кількох ядер.

- L3-кеш – спільний для всіх ядер процесора, працює повільніше, але значно збільшує продуктивність багатоядерних обчислень.

У високопродуктивних системах використовується також L4-кеш, який може бути розташований окремо від процесора та працювати як буфер між кешем нижчого рівня та оперативною пам’яттю.

Використання кеш-пам’яті дозволяє зменшити навантаження на оперативну пам’ять та забезпечує значний приріст продуктивності, особливо у випадку складних обчислень та роботи з великими наборами даних.

Оперативна та постійна пам'ять

Оперативна пам’ять (ОЗП, RAM) є основним сховищем для даних, які активно використовуються операційною системою та запущеними програмами. Її головна особливість – це висока швидкість доступу, яка дозволяє процесору отримувати необхідні дані в найкоротший час.

Проте оперативна пам’ять має одну суттєву особливість – вона енергозалежна, тобто всі дані в ній зникають після вимкнення комп’ютера. Це означає, що для довготривалого зберігання інформації необхідні накопичувачі, такі як HDD або SSD, які можуть зберігати дані навіть без живлення.

Розглянемо розвиток оперативної пам’яті:

1. **SDR SDRAM** (Single Data Rate Synchronous DRAM) – це перше покоління синхронної динамічної пам’яті, синхронізованої з тактовим сигналом системної шини. Передача даних відбувається тільки на одному фронті такту. Основні характеристики:

Частоти: 66–133 МГц.

Напруга: 3.3 В.

Пропускна здатність: до 1.06 ГБ/с.

Особливість: синхронна взаємодія з процесором (на відміну від асинхронної DRAM), що підвищило ефективність обміну даними.

2. **DDR SDRAM** (DDR1) – перший крок до подвоєння продуктивності – передача даних на обох фронтах тактового сигналу (тому ефективна швидкість (у МТ/с) = тактова частота × 2). Основні характеристики:

Частоти: 200–400 МГц (ефективна швидкість 400–800 МТ/с).

Напруга: 2.5 В.

Пропускна здатність: до 3.2 ГБ/с.

Використання: початок 2000-х.

3. **DDR2 SDRAM** – покращена архітектура з prefetch ×4 і меншим енергоспоживанням. Основні характеристики:

Частоти: 400–800 МГц (ефективна швидкість 800–1600 МТ/с).

Напруга: 1.8 В.

Пропускна здатність: до 12.8 ГБ/с.

4. **DDR3 SDRAM** – збільшено prefetch до ×8, підвищено щільність і знижено споживання. Основні характеристики:

Частоти: 800–2133 МГц (ефективно 1600–4266 МТ/с).

Напруга: 1.5 В.

Пропускна здатність: до 34 ГБ/с.

5. **DDR4 SDRAM** – суттєво оновлена структура, більша стабільність і енергоефективність. Основні характеристики:

Частоти: 1600–3200 МГц (ефективно 3200–6400 МТ/с).

Напруга: 1.2 В.

Пропускна здатність: до 51.2 ГБ/с.

6. **DDR5 SDRAM –** найновіше покоління з подвійними каналами на модуль (2×32 біти), вбудованим контролером живлення (PMIC) та ECC навіть у споживчих версіях. Основні характеристики:

Частоти: 4800–8400 МГц (ефективно до 16800 МТ/с).

Напруга: 1.1 В.

Пропускна здатність: 38–128 ГБ/с.

7. **LPDDR** (Low Power DDR) – енергоефективна версія DDR для мобільних пристроїв. Має меншу напругу, оптимізований режим сну й активне керування живленням. Основні покоління та характеристики:

LPDDR1–LPDDR2: ранні покоління (смартфони 2008–2013 рр.), ефективні частоти до 1066 МТ/с.

LPDDR3: 2133 МТ/с, 1.2 В, використовується у планшетах і ноутбуках.

LPDDR4/4X: двоканальна архітектура, частоти до 4266 МТ/с, напруга 1.1–0.6 В.

LPDDR5/5X: до 8533 МТ/с, зменшення енергоспоживання до 30%, застосовується в ARM-ноутбуках і топових смартфонах.

8. **GDDR** (Graphics DDR) – спеціалізована пам’ять для графічних процесорів. Характеризується широкою шиною даних і надвисокими частотами. Основні покоління та характеристики:

GDDR3: близько 8–16 ГБ/с, застосовувалася у GPU NVIDIA 8000 серії.

GDDR5: 5–7 ГГц (ефективно до 28 ГБ/с на лінію).

GDDR6: до 18 Гбіт/с на контакт, понад 500 ГБ/с загальної пропускної здатності.

GDDR6X: покращене кодування PAM4, до 1 ТБ/с у сучасних RTX-відеокартах.

9. **HBM** (High Bandwidth Memory) – розроблена для високопродуктивних систем (GPU, HPC, AI). Це 3D-стекована пам’ять, де кристали DRAM з’єднані вертикально через TSV (Through-Silicon Vias, рис.4.17). Особливістю є низька затримка, надзвичайно висока пропускна здатність, але висока вартість. Використовується зазвичай для серверів та прискорювачів ШІ (NVIDIA Hopper, AMD Instinct, Intel Gaudi).Основні покоління та характеристики:

HBM1: 4 стеки × 128 ГБ/с → ~512 ГБ/с.

HBM2: до 2 ТБ/с.

HBM3 / HBM3E: понад 3 ТБ/с при частотах 6–9 Гбіт/с на контакт.

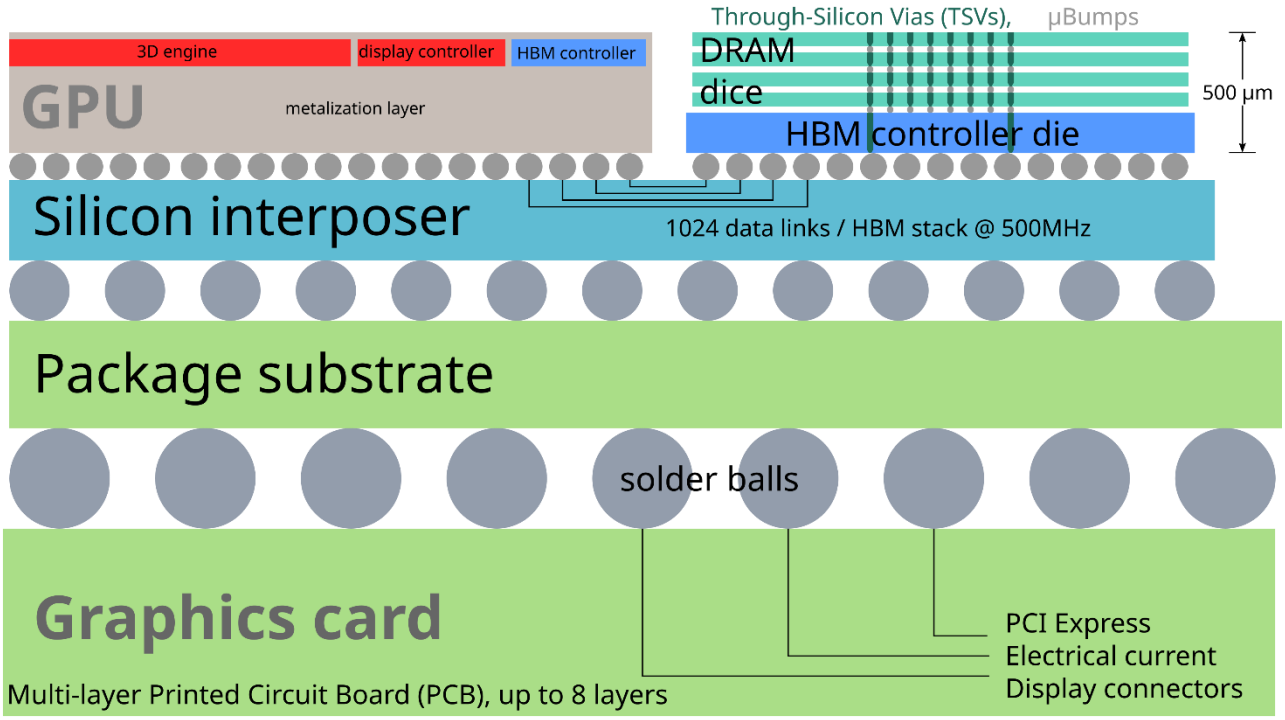

Рисунок 4.17 – Схематичний переріз відеокарти, що використовує HBM [26]

Постійна пам’ять (HDD, SSD) використовується для довготривалого зберігання операційної системи, програм та файлів користувача. Вона значно повільніша, ніж ОЗП, але дозволяє зберігати великі обсяги даних.

Типами постійної пам’яті є:

- **SSD (Solid State Drive)** – твердотільний накопичувач, забезпечує достатньо швидкий доступ до пам’яті, але має обмежену кількість циклів перезапису і відносно низьку надійність при тривалому зберіганні без живлення.
- **HDD (Hard Disk Drive)** – магнітний жорсткий диск, що має велику ємність та відносно низьку швидкодію у порівнянні з SSD. Є більш надійним для довготривалого зберігання, але дуже чутливим до механічних впливів та пошкоджень.

- **SSHD (Solid State Hybrid Drive)** – гібридний накопичувач, що поєднує в собі жорсткий диск (HDD) великої ємності та невеликий обсяг твердотільної пам’яті (SSD), що використовується як кеш для зберігання найчастіше використовуваних даних і програм, що забезпечує швидший доступ у порівнянні зі звичайним HDD.

Щоб оптимізувати роботу системи, сучасні операційні системи використовують віртуальну пам’ять, яка дозволяє частину дискового простору використовувати як додаткову оперативну пам’ять (файл підкачки або swap). Проте швидкість доступу до віртуальної пам’яті є значно нижчою, ніж до фізичної оперативної пам’яті.

З розвитком комп’ютерних технологій постійно вдосконалюються стандарти і постійної пам’яті, що дозволяє підвищити швидкість роботи комп’ютерних систем.

Для з’єднання компонентів пам’яті з материнською платою використовується швидкісний інтерфейс PCI Express (PCIe). Він дозволяє передавати дані на дуже високій швидкості, що особливо важливо для твердотільних накопичувачів через інтерфейс NVMe (Non-Volatile Memory Express).

Твердотільні накопичувачі з інтерфейсом NVMe є сучасною альтернативою традиційним SSD, які працюють через повільніший інтерфейс SATA. Завдяки прямому підключенню до PCIe, накопичувачі можуть досягати швидкостей читання та запису у 10-15 разів вищих, ніж у звичайних SSD. Це значно прискорює завантаження операційної системи, запуск програм та обробку великих обсягів даних.

Особливості архітектури GPU.

Графічний процесор (GPU, Graphics Processing Unit) є спеціалізованим обчислювальним пристроєм, розробленим для обробки графічних даних. Проте за останні десятиліття його функціональність значно розширилася, і сьогодні GPU використовується не тільки для рендерингу зображень, але й для високопродуктивних обчислень у наукових дослідженнях, машинному навчанні, криптографії, аналізі великих даних та моделюванні фізичних процесів.

Архітектура GPU принципово відрізняється від архітектури центрального процесора (CPU). У той час як CPU оптимізований для послідовних обчислень і складної логіки, GPU спеціалізується на масовій паралельній обробці великої кількості однотипних операцій. Це дозволяє графічним процесорам виконувати величезну кількість розрахунків одночасно, що особливо важливо для рендерингу відео, моделювання 3D-сцен та нейронних мереж.

Графічний процесор складається з тисяч малих обчислювальних ядер, які можуть працювати паралельно, виконуючи однакові або подібні операції над різними наборами даних. Це основна відмінність від CPU, який має невелику

кількість продуктивних, але універсальних ядер, оптимізованих для послідовного виконання команд. Ключовими особливостями GPU є:

1. Багатопотокова обробка даних – кожне ядро GPU виконує свою частину роботи одночасно, що значно прискорює виконання операцій, які можуть бути розбиті на незалежні частини.

2. Висока пропускна здатність пам’яті – GPU використовує спеціалізовану відеопам’ять (GDDR або HBM), яка має значно вищу пропускну здатність порівняно з оперативною пам’яттю CPU.

3. Відсутність складної логіки керування потоками – на відміну від CPU, GPU не має потужного модуля управління потоками, що дозволяє використовувати більшу частину транзисторів для обчислень.

4. Оптимізація для обробки матричних операцій – графічні процесори чудово підходять для операцій над великими матрицями, що робить їх незамінними в машинному навчанні та AI-обчисленнях.

5. GPU побудований за блоковою модульною архітектурою, де кожен обчислювальний блок складається з багатьох потокових мультипроцесорів (SM – Streaming Multiprocessors), які, у свою чергу, містять сотні або навіть тисячі ядер.

Одна з ключових характеристик графічного процесора, це тип і обсяг відеопам’яті **VRAM** (Video RAM – загальний термін для позначення будь-якого типу пам’яті, використовуваної в GPU), яка використовується для зберігання текстур, моделей, кадрів рендерингу та інших графічних даних. Об’єм відеопам’яті є критично важливим для роботи з високоякісними текстурами у відеоіграх, професійного відеомонтажу та обробки великих масивів даних у машинному навчанні.

Окрім рендерингу графіки, сучасні GPU широко використовуються для виконання загальних обчислювальних задач, що отримало назву GPGPU (General-Purpose computing on Graphics Processing Units).

Серед найбільш поширених застосувань GPGPU можна виділити:

- Штучний інтелект та машинне навчання – GPU використовуються для тренування нейронних мереж завдяки здатності швидко виконувати матричні операції.
- Наукові дослідження та симуляції – використання GPU для моделювання фізичних процесів, біологічних систем, кліматичних змін.
- Криптографія та блокчейн – GPU виконують складні математичні розрахунки для майнінгу криптовалют та шифрування даних.
- Високопродуктивні обчислення (HPC) – GPU застосовуються в суперкомп’ютерах для обчислення складних задач у сфері аеродинаміки, медицини та квантової фізики.
- Технології, такі як CUDA (від NVIDIA) та OpenCL (від AMD, Intel), дозволяють використовувати графічні процесори для загальних обчислень, що значно розширює їх сферу застосування.

Сучасні тенденції розвитку GPU зосереджені на збільшенні енергоефективності, підтримці штучного інтелекту та покращенні швидкодії для обробки великих масивів даних. Технологічний розвиток у сфері графічних процесорів включає: гібридні процесори (APU, Apple M-серія, AMD Ryzen с iGPU), що полягає в поєднанні CPU та GPU на одному кристалі для підвищення продуктивності та енергоефективності; розширена підтримка AI та машинного навчання так новітні GPU, такі як NVIDIA RTX 40xx та 50xx, мають тензорні ядра, оптимізовані для глибокого навчання та обробки нейронних мереж; оптимізація під Ray Tracing – трасування променів у реальному часі для досягнення фотореалістичної графіки, а застосування нових типів пам’яті HBM3, GDDR7 дозволяє збільшити пропускну здатність для обробки графіки та великих обчислювальних задач. Завдяки цим досягненням GPU поступово стають не лише потужними графічними процесорами, але й універсальними обчислювальними прискорювачами.

Взаємодія компонентів через шини.

Будь-яка комп’ютерна система складається з численних апаратних компонентів – процесора, оперативної пам’яті, відеокарти, накопичувачів, мережевих і периферійних пристроїв. Для їхньої ефективної взаємодії необхідний швидкий і надійний спосіб передачі даних. Цю роль виконують системні шини, які забезпечують передачу команд, даних і сигналів управління між компонентами комп’ютера.

У найзагальнішому вигляді шина – це набір провідників або бездротових каналів, які дозволяють компонентам обмінюватися інформацією за певним протоколом. Вона може передавати дані як послідовним, так і паралельним способом, що визначає її продуктивність та ефективність.

У міру розвитку комп’ютерної техніки архітектура шин зазнавала значних змін: від простих паралельних шин, що використовувалися у ранніх комп’ютерах, до сучасних високошвидкісних шин, які підтримують обмін даними на швидкості десятків гігабайтів за секунду.

Шини в комп’ютері можна поділити на три основні категорії залежно від їхнього призначення:

- Шина процесора (системна шина) – забезпечує обмін даними між центральним процесором (CPU) і оперативною пам’яттю (RAM).
- Шина введення/виведення – використовується для підключення периферійних пристроїв, таких як накопичувачі, відеокарти, мережеві адаптери та інші компоненти.
- Шина живлення – відповідає за подачу електроживлення до компонентів комп’ютера.

Найважливішою є системна шина, яка безпосередньо впливає на продуктивність усієї системи, оскільки через неї проходять всі обчислювальні операції.

Системна шина – це головний канал обміну даними між основними компонентами комп’ютера: процесором, оперативною пам’яттю, контролерами вводу-виводу та іншими пристроями. Вона забезпечує спільний доступ до спільної пам’яті та координує передачу команд і даних. У класичних архітектурах (зокрема, фон Неймана) усі пристрої під’єднані до спільної системної шини, а передача інформації здійснюється за принципом "один говорить – усі слухають".

Зазвичай системна шина складається з трьох логічно відокремлених частин:

- **Шина даних (Data Bus)**, що передає дані між процесором, пам’яттю й іншими пристроями. Ширина шини визначає, скільки біт даних може бути передано за один такт (наприклад, 32 або 64 біти), відповідно, більша ширина визначає більшу пропускна здатність.

- **Шина адрес (Address Bus)**, що передає адресу комірки пам’яті або порту пристрою, до якого звертається процесор. Ширина шини адрес визначає максимальний обсяг адресованої пам’яті, так, наприклад, 32-розрядна адресна шина дозволяє звертатися до 2^{32} = 4 ГБ пам’яті.

- **Шина керування (Control Bus)**, що передає сигнали синхронізації та керування, зокрема, сигнали читання, запису, переривання, підтвердження, тощо. Відповідно ця шина визначає, коли і який пристрій має право користуватися системною шиною.

Типи системних шин:

Front Side Bus (FSB) – класична шина між процесором і північним мостом (контролером пам’яті). Використовувалася у процесорах Intel до архітектури Core i-серії.

HyperTransport (AMD) – високошвидкісна двонаправлена шина, що замінила FSB у процесорах AMD.

QuickPath Interconnect (QPI) – аналогічна шина від Intel для з’єднання процесорів і контролерів пам’яті.

DMI (Direct Media Interface) – сучасний інтерфейс Intel, який з’єднує процесор із чипсетом.

NVLink, Infinity Fabric, CXL – високошвидкісні шини нового покоління розроблені для з’єднання процесорів, GPU і пам’яті у гетерогенних системах.

Основні характеристики системної шини: ширина (біт) – визначає обсяг даних, що передається одночасно; частота (МГц або ГГц) – кількість тактових імпульсів за секунду; пропускна здатність (Б/с) = (ширина / 8) × частота; та тип зв’язку: паралельний або послідовний.

У новітніх архітектурах (наприклад, AMD Ryzen або Intel Alder Lake) класичної спільної шини більше немає. Замість неї використовується структура точка-точка (point-to-point), коли процесор має власні контролери пам’яті, PCIe, USB, SATA тощо. Така організація нівелює “вузьке місце” шини так підвищує продуктивність, а також дозволяє масштабувати систему.

Шини введення/виведення.

Для з’єднання внутрішніх і зовнішніх компонентів комп’ютера використовується кілька основних стандартів шин, кожен з яких має свої особливості та сфери застосування.

1. PCI та PCI Express (PCIe) для високопродуктивних блоків.

PCI (Peripheral Component Interconnect) – це застаріла технологія, яка використовувалася для підключення відеокарт, звукових карт, мережевих адаптерів та інших компонентів. Згодом її замінила значно швидша PCI Express (PCIe), яка використовує послідовну передачу даних і підтримує різну кількість ліній зв’язку (x1, x4, x8, x16).

Останні версії PCIe (наприклад, PCIe 4.0, PCIe 5.0) забезпечують пропускну здатність у десятки гігабайтів за секунду, що критично важливо для високопродуктивних відеокарт, SSD-накопичувачів і серверних рішень.

2. SATA та NVMe для більш повільних блоків та накопичувачів.

Для підключення жорстких дисків (HDD) і твердотільних накопичувачів (SSD) історично використовувався інтерфейс SATA (Serial ATA). Він забезпечував достатню пропускну здатність для традиційних HDD, але виявився надто повільним для сучасних SSD.

Новий стандарт NVMe (Non-Volatile Memory Express), який використовує шину PCIe, забезпечує у 5-10 разів вищу швидкість передачі даних, що значно прискорює завантаження операційних систем, запуск програм і обробку великих масивів даних.

3. USB – універсальна шина для периферійних пристроїв.

Для підключення зовнішніх пристроїв використовується USB (Universal Serial Bus). Його останні версії (USB 3.2, USB4) підтримують швидкість передачі даних до 40 Гбіт/с, що робить цей інтерфейс придатним не лише для підключення миші та клавіатури, але й для зовнішніх накопичувачів, дисплеїв та навіть відеокарт.

Аналогами USB є:

RS-232 (COM-порт) – класичний асинхронний послідовний інтерфейс. Використовувався для модемів, принтерів, мишей.

PS/2 – інтерфейс клавіатури й миші.

IEEE 1284 (Parallel Port, LPT) – паралельна шина для принтерів і сканерів.

eSATA (External SATA) – використовувався для зовнішніх накопичувачів.

FireWire (IEEE 1394) – розробка Apple і Sony для передачі мультимедіа (камери, аудіоінтерфейси). Мав швидкість до 3.2 Гбіт/с (IEEE 1394c) та апаратну підтримку peer-to-peer підключення. Витіснений USB 3.0.

USB4 / USB-C – новітнє покоління самої USB-шини, побудоване на базі Thunderbolt 3. Має пропускну здатність до 40–80 Гбіт/с та є єдиним фізичним стандартом для живлення, відео і даних.

Thunderbolt (Intel + Apple) – поєднує PCI Express та DisplayPort в одному кабелі. Має пропускну здатність: до 80 Гбіт/с (Thunderbolt 5, 2024 р.), підтримує живлення (Power Delivery) і підключення ланцюжком (Daisy Chain). Повністю сумісний із USB4.

Отже, з кожним новим поколінням комп’ютерної техніки вимоги до швидкості обміну даними постійно зростають. У зв’язку з цим сучасні системні шини розвиваються у наступних напрямках: збільшення пропускної здатності, зниження затримок та підвищення енергоефективності. Так нові шини оптимізуються для максимально швидкого обміну даними без зайвого споживання енергії, що важливо як для мобільних пристроїв так і для дата-центрів.

Швидкість шини напряму впливає на продуктивність комп’ютера, тому вибір правильного інтерфейсу для підключення компонентів є критично важливим під час збирання або оновлення системи.

Системи охолодження для високопродуктивних пристроїв.

Зі зростанням продуктивності комп’ютерних систем та підвищенням їхніх обчислювальних можливостей виникає серйозна проблема – відведення тепла. Високопродуктивні процесори, відеокарти, сервери та спеціалізовані обчислювальні системи споживають значну кількість енергії, що супроводжується виділенням великої кількості тепла. Якщо це тепло не відводити вчасно, можливий перегрів, що може призвести до зниження продуктивності, нестабільної роботи або навіть пошкодження апаратного забезпечення.

Щоб запобігти цим проблемам, застосовуються різні системи охолодження, які забезпечують ефективний теплообмін та підтримують безпечну температуру роботи компонентів. У сучасних комп’ютерах використовуються повітряне, рідинне та гібридне охолодження, кожен із методів має свої переваги та недоліки.

Оскільки процесори та відеокарти працюють із дуже високими тактовими частотами, виконуючи мільярди операцій на секунду, це створює високе навантаження на електричні схеми, та супроводжується значним тепловиділенням. При перегріві спрацьовують механізми захисту, такі як теплове зниження продуктивності (thermal throttling) – коли система автоматично зменшує тактову частоту, щоб знизити температуру. Це призводить до падіння швидкодії та продуктивності, що є критичним для серверних рішень, систем обробки графічного та аудіо контенту, відео, графічних та аудіо редакторів, ігор та інших ресурсоємних задач. Окрім зниження продуктивності, постійний перегрів може скоротити термін служби компонентів через деградацію матеріалів. Тому ефективне охолодження – це не просто покращення продуктивності, а й запорука довговічності комп’ютерних систем та компонентів.

Розглянемо види систем охолодження (рис. 4.18).

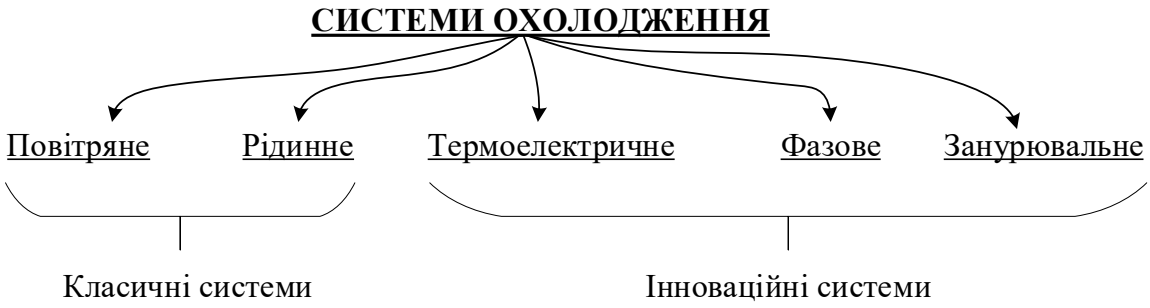

Рисунок 4.18 – Класифікація систем охолодження

Повітряне охолодження. Повітряне охолодження є найбільш поширеним і бюджетним методом, який використовує радіатори та вентилятори для розсіювання тепла. Процес виглядає так:

1. Тепло від процесора або відеокарти передається на радіатор (зазвичай зроблений з алюмінію або міді).

2. Вентилятор охолоджує радіатор, розсіюючи нагріте повітря.

3. Нагріте повітря викидається назовні через корпусні вентилятори.

Типи повітряних систем:

- Звичайні кулери (Stock Coolers) – постачаються разом із процесором, але мають обмежену ефективність.
- Баштові кулери (Tower Coolers) – оснащені великим радіатором і потужними вентиляторами, що дозволяє ефективніше відводити тепло.
- Топ-даун кулери (Low-Profile Coolers) – використовуються у компактних системах, де немає місця для великих радіаторів.

Повітряне охолодження добре підходить для звичайних користувачів і робочих станцій, але для екстремального розгону процесорів або високопродуктивних серверів можуть знадобитися більш ефективні методи.

Рідинне охолодження. Рідинне охолодження використовує замкнутий контур з охолоджувальною рідиною, яка циркулює через систему та переносить тепло від гарячих компонентів до радіатора, де воно розсіюється за допомогою вентиляторів.

Видами рідинного охолодження є

- Замкнуті All-in-One (AIO) системи. Це популярне рішення, яке не потребує додаткового обслуговування. AIO-охолоджувачі продаються у готових комплектах та легко встановлюються.
- Кастомні системи рідинного охолодження, що використовуються ентузіастами для максимальної ефективності та кастомізації. Вони можуть охолоджувати не лише процесор, а й відеокарту, материнську плату та навіть оперативну пам’ять.

Рідинне охолодження ідеально підходить для геймерів, ентузіастів розгону та професійних робочих станцій, які працюють із високими навантаженнями.

Інноваційні методи охолодження. З розвитком технологій з’являються нові методи охолодження, які ще більше підвищують ефективність, до основних таких систем можна віднести:

- Термоелектричне охолодження, що використовують ефект Пельтьє (Peltier) – явище, коли електричний струм використовується для перенесення тепла, проте такі системи мають високе енергоспоживання.
- Фазове охолодження (Phase Change Cooling) – подібне до роботи холодильника, може охолоджувати процесор до негативних температур, але такі системи дорогі енергоємні та складні в експлуатації.
- Занурювальне (імерсивне) охолодження (Immersion Cooling) – комп’ютерні компоненти повністю занурюються у спеціальну діелектричну рідину, яка ефективно відводить тепло. Зазвичай такі системи використовується у дата-центрах.

Як можна зрозуміти з недоліків, такі технології поки, що не знайшли широкого застосування у звичайних комп’ютерах, але активно використовуються у серверних і наукових обчислювальних системах.

Отже, ефективне охолодження є критично важливим для стабільної та довговічної роботи високопродуктивних комп’ютерів. Повітряне охолодження залишається найпростішим і найдоступнішим рішенням, тоді як рідинне охолодження забезпечує вищу продуктивність для потужних ПК.

Розвиток нових технологій охолодження дозволяє підвищити продуктивність комп’ютерів та розширити їхні можливості, особливо у сферах геймінгу, серверних рішень, суперкомп’ютерів та наукових досліджень. У майбутньому можна очікувати ще ефективніші та енергоефективніші методи охолодження, що зробить потужні обчислювальні системи ще продуктивнішими та надійнішими.

4.5. Апаратна віртуалізація: технології Intel VT-x, AMD-V

Окремо слід звернути увагу на системи апаратної віртуалізації. Віртуалізація є одним із ключових напрямків розвитку сучасних обчислювальних систем, оскільки вона дозволяє ефективно використовувати апаратні ресурси, підвищувати безпеку, ізолювати середовища та забезпечувати масштабованість інфраструктури.

В основі віртуалізації лежить можливість запуску декількох операційних систем або ізольованих програмних середовищ на одному фізичному сервері чи комп’ютері. Цей підхід використовується в хмарних обчисленнях, дата-центрах, при тестуванні програмного забезпечення, комп’ютерній безпеці та інфраструктурі DevOps.

Апаратна віртуалізація стала можливою завдяки появі спеціалізованих апаратних технологій підтримки віртуалізації, які забезпечують безпосередню роботу віртуальних машин на рівні процесора. Найвідомішими з них є Intel VT-

x та AMD-V, які розширюють можливості програмних гіпервізорів і роблять віртуалізацію більш ефективною.

Віртуалізація загалом поділяється на програмну та апаратну:

- Програмна віртуалізація здійснюється виключно за допомогою програмних засобів (наприклад, VirtualBox, VMware Workstation), що може створювати додаткове навантаження на процесор і знижувати продуктивність.

- Апаратна віртуалізація використовує спеціальні можливості процесора, які дозволяють віртуальним машинам працювати безпосередньо з ресурсами ЦП без необхідності емулювання, що значно підвищує продуктивність.

Апаратна підтримка віртуалізації дозволяє ізолювати віртуальні машини від основної ОС, що забезпечує вищу безпеку, кращу продуктивність та можливість одночасного використання різних операційних систем без значних втрат швидкодії.

Технологія Intel VT-x.

Intel VT-x (Intel Virtualization Technology) – це набір апаратних розширень для процесорів Intel, які забезпечують більш ефективну віртуалізацію. Вперше ця технологія була представлена у 2005 році і з того часу стала стандартом для всіх сучасних процесорів Intel. VT-x дозволяє програмним гіпервізорам (наприклад, VMware, Hyper-V, KVM, VirtualBox) безпосередньо керувати віртуальними машинами, уникаючи необхідності емулювання привілейованих інструкцій. Основними технологіями Intel VT-x є:

- Перехід у режим "root mode" для гіпервізора – забезпечує прямий контроль над віртуальними машинами без необхідності емулювати привілейовані операції.

- VMCS (Virtual Machine Control Structure) – спеціальна структура керування, яка зберігає стан віртуальних машин і керує їхнім виконанням.

- EPT (Extended Page Tables) – покращене управління пам’яттю для віртуальних машин, що зменшує накладні витрати на трансляцію адрес.

- Unrestricted Guest – дозволяє запускати гостові ОС без обмежень, навіть без використання захищеного режиму.

- VT-d (Intel Virtualization for Directed I/O) – розширення для прискорення роботи пристроїв введення/виведення у віртуальному середовищі.

Технологія Intel VT-x широко застосовується в хмарних обчисленнях (AWS, Azure, Google Cloud), в тестуванні та розробці програмного забезпечення (контейнери, віртуальні середовища), при запуску ізольованих ОС для безпеки (наприклад, Windows Sandbox) та в серверних рішеннях для віртуалізації робочих середовищ.

Технологія AMD-V.

AMD-V (AMD Virtualization) – це аналогічна технологія віртуалізації, розроблена AMD. Вона була представлена у 2006 році та має схожі функціональні можливості з Intel VT-x. AMD-V дозволяє зменшити накладні витрати під час виконання віртуальних машин і покращує продуктивність гіпервізорів, таких як KVM, VMware, Hyper-V, Xen. Основними технологіями AMD-V є:

- SVM (Secure Virtual Machine) – захищена архітектура для ізольованого виконання віртуальних машин.

- Rapid Virtualization Indexing (RVI) – аналог Intel EPT, який прискорює трансляцію сторінок пам’яті віртуальних машин.

- Nested Paging – покращене управління пам’яттю, що дозволяє віртуальним машинам працювати майже без втрати продуктивності.

- AMD-Vi (I/O Virtualization) – аналог Intel VT-d, що дозволяє прискорювати обробку введення/виведення у віртуальному середовищі.

Технологія AMD-V використовується в серверних рішеннях AMD EPYC та Ryzen PRO, оптимізоване для віртуалізації великих кластерів у дата-центрах, та широко застосовується в Linux-середовищах для KVM та контейнеризації.

Загалом, обидві технології забезпечують високий рівень продуктивності для віртуалізації, і вибір між ними залежить від конкретних вимог системи та доступного обладнання.

Отже, апаратна віртуалізація – це ключова технологія, яка змінила підхід до використання обчислювальних ресурсів. Завдяки Intel VT-x та AMD-V стало можливим ефективно запускати віртуальні машини, ізолювати середовища, підвищувати безпеку та оптимізувати серверні інфраструктури. В майбутньому роль віртуалізації лише зростатиме, оскільки вона є основою для хмарних технологій, контейнеризації, кібербезпеки та високопродуктивних обчислень.

Висновки до розділу 4

Поділ обчислювальних машин на покоління базується на суттєвих технологічних проривах, які впливали на архітектуру, продуктивність, розмір, енергоспоживання та сферу застосування комп’ютерів. Основні критерії, що визначали перехід між поколіннями: зміна елементної бази (електронні лампи → транзистори → мікросхеми → мікропроцесори → квантові системи); нові архітектурні рішення (від фон Неймана до паралельних та квантових обчислень); розвиток програмного забезпечення (від машинного коду до штучного інтелекту); зміна сфери використання (від військових досліджень до персонального застосування); розвиток засобів зберігання та передачі даних (від перфокарт до хмарних сховищ і квантової пам’яті).

Так, перше покоління комп’ютерів було етапом становлення електронної обчислювальної техніки. Незважаючи на свої недоліки, ці машини

започаткували революцію в інформатиці, заклали основи фон-нейманівської архітектури, дали старт програмуванню та започаткували нову еру автоматизованих обчислень.

Друге покоління стало ключовим етапом в історії комп’ютерів. Перехід від електронних ламп до транзисторів зробив комп’ютери компактнішими, швидшими та надійнішими. З’явилися мови програмування, операційні системи та перші комерційні комп’ютери. Однак для ще більшої мініатюризації та доступності знадобився винахід інтегральних схем, що привів до появи третього покоління обчислювальних машин.

Третє покоління обчислювальних машин стало основою для розвитку сучасних обчислювальних систем. Перехід на інтегральні схеми, поява операційних систем, розвиток програмування та стандартизація архітектури дозволили зробити комп’ютери більш доступними та продуктивними. Наступним кроком було винайдення мікропроцесора, який відкрив еру персональних комп’ютерів.

Четверте покоління стало основою для сучасних обчислювальних систем. Персоналізація комп’ютерів, поява GUI, розвиток програмного забезпечення та мікропроцесорів зробили ПК доступними для всіх. Наступним кроком став перехід до оптимізованих процесорних архітектур, що дали новий поштовх комп’ютерній індустрії та стало новим етапом розвитку обчислювальної техніки завдяки впровадженню RISC-архітектури, багатоядерних процесорів, гетерогенних обчислень та паралельної обробки даних.

Так, кожен етап розвитку комп’ютерних технологій визначався новими потребами та інноваціями, які усували обмеження попередніх поколінь, проте розвиток обчислювальних систем не зупиняється і визначається постійним розвитком та новими технологіями та винаходами. Майбутні покоління комп’ютерної техніки будуть характеризується появою квантових, нейроморфних, оптичних та біологічних обчислювальних систем. Більшість цих технологій вже перебуває в стадії розробки, та мають експериментальні моделі, що демонструють потенціал майбутніх технологій.

Архітектура фон Неймана стала фундаментом розвитку сучасної обчислювальної техніки. Її універсальність, заснована на збереженні програм та даних у спільному просторі пам’яті, забезпечила можливість створення гнучких і багатофункціональних комп’ютерних систем. Незважаючи на появу нових підходів (гарвардська архітектура, багаторівневі кеші, спеціалізовані прискорювачі), принципи, закладені Джоном фон Нейманом, залишаються актуальними й досі. Саме ця концепція визначила подальший шлях розвитку апаратного та програмного забезпечення, зробивши можливим створення універсальних машин, здатних вирішувати широкий спектр завдань.

Визначено і певні фундаментальні недоліки архітектура фон Неймана, пов’язані з низькою пропускною здатністю пам’яті та шини, що уповільнює виконання обчислень, щоб мінімізувати ці проблеми у сучасних системах

використовуються кеш-пам’ять, багатоканальна пам’ять, незалежні шини PCIe та спеціалізовані прискорювачі. Однак навіть ці покращення не повністю усувають вузькі місця пам’яті та шини, тому у майбутньому комп’ютерна індустрія може перейти до принципово нових архітектур, таких як нейроморфні чи квантові обчислення, коли шина пам’яті взагалі не використовується.

Архітектура сучасного персонального комп’ютера частково вирішує проблему вузького місця, використовуючи ієрархічну будову пам’яті та розпаралелювання шини. Також організація таких комп’ютерів базується на модульному підході, що дозволяє легко замінювати та оновлювати компоненти, вдосконалювати продуктивність, впроваджувати швидшу пам’ять, багатоядерні процесори, потужніші GPU та нові системи зберігання даних.

Сучасні мобільні пристрої є потужними, компактними та енергоефективними завдяки високому рівню інтеграції компонентів у форматі System-on-a-Chip (SoC). Вони оптимізовані для автономної роботи, бездротового зв’язку, мультимедіа та штучного інтелекту. Основна тенденція розвитку – збільшення продуктивності при мінімальному енергоспоживанні, впровадження нейромережевих обчислень, гнучких дисплеїв, альтернативних методів живлення. У майбутньому архітектура мобільних пристроїв продовжить розвиватися, наближаючись за продуктивністю до настільних ПК, але залишаючись енергоефективною та портативною.

Мейнфрейми та суперкомп’ютери – це високопродуктивні системи, що розв’язують критично важливі завдання: мейнфрейми забезпечують безперервну роботу бізнес-процесів та управління транзакціями, а суперкомп’ютери використовуються для наукових обчислень, симуляцій та штучного інтелекту. З розвитком технологій обидва типи систем стають ще продуктивнішими, енергоефективнішими та адаптованими для нових викликів цифрової ери.

Процесорна архітектура продовжує розвиватись та змінювати комп’ютерну індустрію. CISC-процесори (x86-64) домінують у ПК та серверах, тоді як RISC (ARM, RISC-V) завойовують мобільний і серверний ринки завдяки енергоефективності. Можна напевне сказати, що майбутнє технологій за багатоядерністю, гібридними архітектурами та інтеграцією штучного інтелекту в процесорні блоки.

Ієрархія пам’яті також є одним з ключових елементів у сучасних комп’ютерних системах, оскільки вона визначає ефективність обробки даних. Так використання кеш-пам’яті дозволяє прискорити доступ до часто використовуваних даних, оперативна пам’ять забезпечує швидку взаємодію з програмами, а накопичувачі HDD та SSD дозволяють довготривале зберігання інформації. Завдяки постійному розвитку стандартів пам’яті сучасні комп’ютери стають ще швидшими, що відкриває нові можливості для високопродуктивних обчислень та штучного інтелекту.

Графічні процесори GPU принципово відрізняється від CPU масовою паралельністю, вони використовують тисячі простих ядер замість кількох

складних, що робить їх використання оптимальним для матричних операцій, рендерингу та машинного навчання.

Системні шини відіграють вирішальну роль у функціонуванні комп’ютера, забезпечуючи ефективний обмін даними між його компонентами. Чим швидше працює шина, тим ефективніше використовуються ресурси системи, а отже – тим вища продуктивність усього пристрою.

Завдяки постійному розвитку стандартів, таких як PCIe, NVMe, USB4, сучасні комп’ютери отримують все більшу швидкість обміну даними, що відкриває можливості для нових технологій, зокрема штучного інтелекту, доповненої реальності та високопродуктивних обчислень. У майбутньому очікується подальше вдосконалення архітектури шин, що зробить комп’ютерні системи ще швидшими та енергоефективнішими.

Апаратна віртуалізація є однією з тих технологій, що допомагають ефективно вирішувати проблему простою ресурсів та швидкості доступу до них, коли обчислювальні потужності процесорів та інших компонентів комп’ютерів розподіляються на віртуальні машини, які паралельно виконують свої задачі, та розподіляють ресурси між незалежними користувачами. Так, саме завдяки апаратній віртуалізації стала можливою сучасна хмарна інфраструктура, де тисячі клієнтів використовують ізольовані віртуальні машини на спільному фізичному обладнанні.

Питання до самоконтролю

1. Від якого латинського слова походить термін «калькулятор» і з яким давньоримським пристроєм це пов'язано?
2. Які основні компоненти аналітичної машини Беббіджа і чому вона вважається прообразом сучасного комп'ютера?
3. Назвіть елементну базу кожного з чотирьох поколінь комп'ютерів та вкажіть по одному представнику кожного покоління.
4. Що таке закон Мура?
5. Яке головне обмеження квантових комп'ютерів на сьогодні?
6. Опишіть основні принципи архітектури фон Неймана. Назвіть основні компоненти які вона включає.
7. У чому полягає проблема «вузького місця» архітектури фон Неймана і які методи її подолання використовуються у сучасних системах?
8. Опишіть цикл «вибірка – декодування – виконання» процесора. Яку роль у ньому відіграють лічильник команд (PC) і регістр інструкцій (IR)?
9. Чим гарвардська архітектура відрізняється від фон-нейманівської і де вона застосовується?
10. Поясніть різницю між CISC і RISC архітектурами.
11. Що таке конвеєрна обробка?
12. В чому полягає різниця між суперконвеєрним і суперскалярним процесором?

13. Що таке MPP-система?

14. У чому перевага кластерної архітектури з точки зору масштабованості і відмовостійкості?

15. Що таке VLIW-архітектура?

16. Що таке Constellation-система?

17. Що таке технологія Hyper-Threading?

18. Опишіть класифікацію Флінна. Наведіть приклади реальних систем для класів SISD, SIMD і MIMD. Чому клас MISD вважається порожнім?

19. Що таке архітектура SoC? Які компоненти вона об'єднує і які переваги та недоліки має порівняно з традиційною платформою «процесор + чипсет»?

20. Опишіть ієрархію пам'яті сучасного комп'ютера. Який принцип покладено в основу її побудови?

21. Чим L1, L2 і L3 кеш відрізняються між собою?

22. Порівняйте типи оперативної пам'яті DDR4, DDR5 і HBM.

23. Чим SSD відрізняється від HDD і в чому перевага NVMe порівняно зі звичайним SATA SSD?

24. У чому різниця між архітектурою CPU і GPU?

25. Що таке GPGPU?

26. Опишіть складові системної шини та їх призначення.

27. Як ширина адресної шини визначає максимальний обсяг адресованої пам'яті?

28. Чим PCIe відрізняється від класичного PCI? Як позначення x1, x4, x8, x16 пов'язані з пропускною здатністю?

29. Порівняйте типи охолодження за ефективністю і сферою застосування.

30. Що таке thermal throttling і чому він знижує продуктивність?

31. Що таке апаратна віртуалізація і чим вона відрізняється від програмної?

32. Що таке тензорні ядра GPU і для яких задач вони оптимізовані порівняно зі звичайними обчислювальними ядрами?

33. Що таке гіпервізор? Чим гіпервізор першого типу (bare-metal) відрізняється від гіпервізора другого типу (hosted)?

34. Що таке EPT (Extended Page Tables) в Intel VT-x?

35. Що таке контейнеризація і чим контейнер відрізняється від віртуальної машини?

РОЗДІЛ 5. ОСНОВИ РОБОТИ З АСЕМБЛЕРОМ

5.1. Асемблер та мова асемблера.

Як ми розглядали раніше, на найнижчому рівні процесор розуміє лише послідовності нулів і одиниць, так званий **машинний код**. Кожна інструкція процесора закодована як двійкове число певної довжини. Наприклад, команда додавання двох регістрів на архітектурі x86 виглядає як:

```
00000011 11000011
```

Бінарний код цієї команди складається з двох частин (байтів):

1. Перший байт (00000011 або 03 у шістнадцятковій системі) – це «Opcode» (код операції). В архітектурі x86 код 03 означає операцію додавання (ADD), яка вказує процесору, що потрібно додати значення одного операнда до іншого, причому місце призначення та джерело (яке значення додається до якого і куди потім зберігається результат) визначається наступним байтом.

2. Другий байт (11000011 або C3 у шістнадцятковій системі) – це байт «ModR/M» спеціальний байт у машинному коді архітектури x86, який, як ми вже зазначили, іде одразу після коду операції (Opcode), та «пояснює» процесору, з якими саме даними працювати: 11 (перші два біти) визначає, що обидва операнди є регістрами, а 000 та 011 (наступні дві частинки по три біти) – вказують на конкретні регістри EAX та EBX відповідно.

Як можна побачити з такого просто прикладу, писати програми безпосередньо у машинному коді практично неможливо, людина не здатна сприймати і відтворювати такі послідовності без помилок. Для вирішення цієї проблеми розроблюється спеціальна мова програмування, яка перетворює набір бітів інструкцій в відносно зручні для опрацювання символьні представлення машинного коду – **мову асемблера**, у якій кожній машинній інструкції відповідає коротке текстове позначення – **мнемоніка**. Наприклад, розгляненена вище команда додавання записується як:

```
ADD EAX, EBX
```

Ця інструкція може читатись як «додати вміст регістра EBX до регістра EAX». Це вже зрозуміло людині, хоча і потребує знання архітектури процесора.

Важливо не плутати два поняття: мову асемблера (текстовий запис програми) і сам асемблер (програма, що транслює цей текстовий запис в машинний код), оскільки в побутовому мовленні обидва поняття часто називають просто «асемблером», що є технічно неточним, але загальноприйнятим:

- **Мова асемблера (Assembly language)** – це низькорівнева мова програмування, де замість двійкових кодів (0 та 1) використовуються короткі слова – мнемоніки (наприклад, MOV, ADD, PUSH). Вона дозволяє людині писати зрозумілий код, який майже один-в-один відповідає командам процесора.

- **Асемблер (Assembler)** – це спеціальна програма-транслятор (утиліта), яка перетворює написаний текст інструкцій (мовою асемблера) у машинний код (набір бітів), зрозумілий комп'ютеру (рис. 5.1).

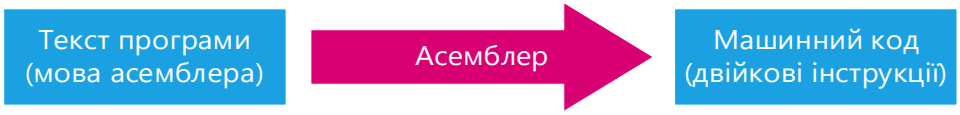

Рисунок 5.1 – Принцип роботи Асемблера.

Мови програмування прийнято поділяти за рівнем абстракції тому виділимо місце асемблера за допомогою таких рівнів (рис.5.2). Мови високого рівня абстрагують програміста від деталей апаратного забезпечення – один рядок коду на Python може перетворитись на десятки машинних інструкцій. Асемблер навпаки – один рядок відповідає рівно одній машинній інструкції, що дає програмісту повний контроль над тим, що відбувається на апаратному рівні.

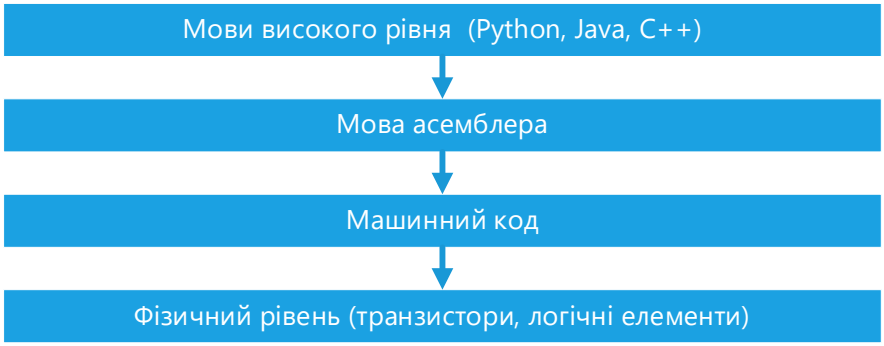

Рисунок 5.2 – Місце асемблера в ієрархії програмування.

Для чого використовується асемблер сьогодні? З появою ефективних компіляторів мов високого рівня асемблер перестав використовуватись для написання повноцінних програм. Однак він залишається незамінним у кількох областях:

Системне програмування – ядра операційних систем містять фрагменти на асемблері для операцій які неможливо виразити мовою С: перемикання контексту між процесами, ініціалізація процесора при завантаженні, управління привілейованими регістрами.

Вбудовані системи – мікроконтролери з обмеженими ресурсами (кілька кілобайт пам'яті, відсутність операційної системи) часто програмуються на асемблері для максимальної ефективності використання ресурсів.

Оптимізація критичних ділянок – компілятори не завжди генерують оптимальний код. У задачах де критична швидкодія – обробка сигналів, відеокодеки, криптографічні алгоритми – програмісти вручну пишуть асемблерні вставки.

Реверс-інжиніринг та кібербезпека – аналіз шкідливого програмного забезпечення, пошук вразливостей, дослідження закритого програмного забезпечення виконується через дизасемблювання – зворотне перетворення машинного коду в асемблер.

Навчання – вивчення асемблера дає глибоке розуміння того як працює процесор, як керується пам'ять, як виконуються програми.

Принципова відмінність асемблера від мов високого рівня – він не є переносимим. Програма написана для процесора архітектури x86 не працюватиме на ARM або RISC-V без повного переписування. Кожна архітектура має власний набір інструкцій, власні регістри і власний синтаксис асемблера. Саме тому мови високого рівня витіснили асемблер у більшості застосувань – програма на C компілюється під будь-яку архітектуру без змін у вихідному коді.

На сьогодні архітектура IA-32 (x86) залишається стандартом сумісності для більшості пристроїв, тому в посібнику розглядається асемблер саме для цієї архітектури. Для вивчення основ програмування мовою асемблера зручним інструментом є середовище Microsoft Visual Studio, яке дозволяє інтегрувати асемблерний код безпосередньо у програму написану мовою C за допомогою блоку `__asm`. Такий підхід не потребує налаштування зовнішніх компіляторів чи окремих асемблерних файлів і дозволяє зосередитись на вивченні самої мови асемблера, не відволікаючись на організаційні питання середовища розробки.

> NB! Для використання асемблерних вставок у Visual Studio необхідно створити проєкт типу Win32 (x86), оскільки 64-бітні проєкти (x64) не підтримують блок `__asm`.

5.2. Регістри процесора та їх структура

Для того щоб писати програми мовою асемблера, недостатньо знати лише синтаксис інструкцій – необхідно розуміти з чим саме ці інструкції працюють. Основним робочим простором процесора є регістри – невеликі комірки пам'яті вбудовані безпосередньо в процесор, в яких зберігаються дані під час виконання обчислень. Саме регістри є операндами більшості інструкцій асемблера, тому перш ніж перейти до програмування, необхідно розглянути їх організацію та призначення.

Регістр – це надшвидка комірка пам'яті розташована безпосередньо всередині процесора. На відміну від оперативної пам'яті доступ до регістра відбувається за один такт процесора без жодних затримок. Архітектура IA-32 має кілька груп користувацьких регістрів кожна з яких призначена для певного класу задач:

- Регістри загального призначення;
- Лічильник команд;

- Регістр прапорців;
- Сегментні регістри;
- Регістри математичного співпроцесора x87.

Регістри загального призначення

До регістрів загального призначення належать регістри, які фізично розміщені всередині ALU, тому їх інколи називають регістрами ALU. Це вісім 32-бітних регістрів, що використовуються для зберігання цілих чисел, адрес і проміжних результатів обчислень (рис.5.3). Усі ці регістри доступні для зберігання операндів без особливих обмежень, хоча в деяких випадках деякі з них мають певне функціональне призначення. Особливістю регістрів загального призначення є те, що вони містять у собі молодші 16-бітні і 8-бітні частини до яких можна звертатись окремо. Звернення здійснюється за їх іменами. Важливо зазначити, що використовувати для самостійної адресації можна тільки молодші 16- та 8-розрядні частини цих регістрів. Старші 16 бітів для самостійної роботи не доступні. Регістри ESI, EDI, ESP і EBP не мають окремих 8-бітних частин – лише 16-бітні (SI, DI, SP, BP).

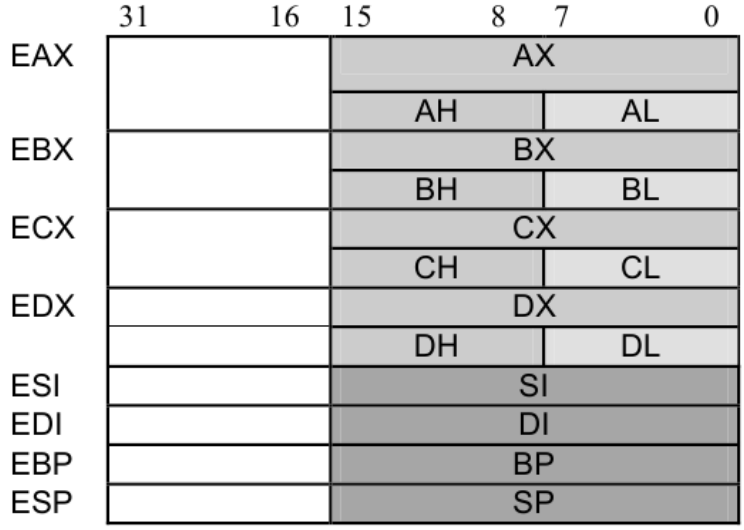

Рисунок 5.3 – Регістри загального призначення

- **EAX/AX/AH/AL** (Accumulator register) – регістр-акумулятор, застосовується для збереження результатів проміжних операцій, використовується у всіх операціях введення-виведення, в арифметичних операціях, у деяких командах його використання обов’язкове;

- **EBX/BX/BH/BL** (Base register) – базовий регістр, застосовується для збереження базової адреси деякого об’єкта в пам’яті (єдиний, який використовується в індексній адресації), може використовуватись при розрахунках для зберігання проміжних результатів;

- **ECX/CX/CH/CL** (Count register) – регістр-лічильник, використовується як лічильник циклів та елементів при виконанні строкових операцій, може використовуватись у розрахунках для зберігання проміжних результатів;

- **EDX/DX/DH/DL** (Data register) – регістр даних так само, як і регістр EAX/AX/AH/AL призначений для зберігання проміжних даних, у деяких командах його пряме використання обов’язкове, а в деяких він використовується неявно.

- **ESI/SI** (source index register) – регістр індексу джерела, містить адресу даних, які розміщуються в сегменті, що адресується регістром DS, а в ланцюгових операціях містить поточну адресу елемента в ланцюгу джерелі;

- **EDI/DI** (destination index register) – регістр індексу приймача, адресує дані, які розміщуються в сегменті, базова адреса якого знаходиться в регістрі ES, також може містити зміщення рядка-приймача при виконанні строкових операцій.

В архітектурі мікропроцесора на програмно-апаратному рівні підтримується така структура даних як стек. Для роботи зі стеком існують спеціальні регістри:

- **ESP/SP** (Stack Pointer register) – регістр покажчика стеку, який містить покажчик на верхівку стеку в поточному сегменті стеку;

- **EBP/BP** (Base Pointer register) – регістр покажчика бази кадру стеку призначений для організації довільного доступу до даних всередині стеку.

Лічильник команд EIP (Extended Instruction Pointer)

Містить адресу наступної інструкції яку виконає процесор. Програміст не може змінити EIP безпосередньо – він змінюється автоматично після кожної інструкції, або командами переходу (JMP, CALL) і поверненням з підпрограми (RET). Розуміння EIP є ключовим для розуміння того як процесор виконує програму послідовно і як організовуються переходи.

Асемблер виконує два проходи по вихідному тексту програми:

За перший прохід асемблер читає весь текст програми і будує таблицю символів – список усіх міток і відповідних їм адрес. Це необхідно тому, що програма може містити перехід на мітку яка визначена нижче і асемблер не може знати її адресу не прочитавши весь текст програми наперед:

```
asm
    JMP end          ; перехід на мітку "end", але асемблер не знає де вона
    MOV EAX, 0
end:                 ; ось вона, але асемблер дізнається про це лише тут
```

Під час другого проходу асемблер знову читає текст програми і тепер генерує машинний код, підставляючи реальні адреси з таблиці символів замість символічних міток.

Регістр прапорців EFLAGS

Розрядність регістра прапорців EFLAGS/FLAGS (flag register) дорівнює 32(16) біт. Окремі розряди цього регістра мають своє функціональне призначення й називаються прапорцями. Молодша частина регістра EFLAGS повністю відповідає регістру FLAGS процесора i8086. Детально регістр EFLAGS буде розглядатись у наступному підрозділі.

Сегментні регістри

Будь-яка програма складається з трьох сегментів: коду, даних та стеку. Логічно машинні команди в архітектурі IA-32 побудовані так, що при виборі кожної команди для доступу до даних програми або стеку неявно використовується інформація, яка розміщена у визначених сегментних регістрах. Залежно від режиму роботи процесора за їх вмістом визначаються адреси пам’яті, з яких починаються відповідні сегменти. У програмній моделі IA-32 є шість 16-бітних регістрів CS, SS, DS, ES, GS та FS, які призначені для сегментації пам'яті – механізму організації адресного простору доступу до чотирьох типів сегментів. Призначення сегментних регістрів такі:

- регістр CS (Code Segment – сегмент коду) – відповідає сегменту команд, що виконуються в даний момент;

- регістр DS (Data Segment – сегмент даних) – відповідає сегменту даних, з якими працює процесор;

- регістр ES (Extra Segment – додатковий сегмент) – відповідає додатковому сегменту даних;

- регістр SS (Stack Segment – сегмент стека) – відповідає сегменту стека.

- регістри FS та GS (F та G – наступні після E літери) – додаткові сегменти, що можуть використовуватись операційною системою комп’ютера

Сегмент коду містить команди програми. Для доступу до цього сегмента служить регістр сегмента коду CS (Code segment register). Він містить адресу сегмента з машинними командами, до якого має доступ процесор (ці команди завантажуються в конвеєр процесора).

Сегмент даних містить дані, що обробляються програмою. Для доступу до цього сегмента служить регістр сегмента даних DS (Data segment register), у якому зберігається адреса сегмента даних поточної програми.

Сегмент стеку представляє собою область пам’яті, яка називається стеком. Роботу зі стеком процесор організовує за таким принципом: останній записаний у цю область пам’яті елемент вибирається першим. Для доступу до цієї області

призначений регістр сегмента стеку SS (Stack segment register), який містить адресу сегмента стеку.

Додатковий сегмент даних. Неявно алгоритми виконання більшості машинних команд передбачають, що дані, які ними обробляються, розташовані в одному сегменті даних, адреса якого розміщена в регістрі сегмента даних DS. Якщо програмі недостатньо одного сегмента даних, то вона має можливість задіяти ще три додаткових сегменти даних. Але на відміну від основного сегмента даних, адреса якого розміщується в DS, при використанні додаткових сегментів даних їх адреси потрібно вказувати явно за допомогою спеціальних префіксів перевизначення сегментів у команді. Адреси додаткових сегментів повинні міститися в регістрах додаткового сегмента даних ES (Extension Data Segment register), GS, FS.

Також важливо зазначити, що у сучасних 32-бітних операційних системах (Windows, Linux) використовується **плоска модель пам'яті** (flat memory model) де всі сегментні регістри вказують на один і той самий адресний простір.

Регістри співпроцесора x87

Окремий стек з восьми 80-бітних регістрів ST(0)…ST(7) призначений написання програм, що використовуються для складних математичних обчислень та використовують типи даних призначені для чисел з плаваючою комою. Детально робота з цими регістрами розглядається у підрозділі про співпроцесор x87.

Системні регістри

Окремо розглянемо системні регістри – це спеціальні регістри процесора, що керують роботою самого процесора, організацією пам'яті і режимами його функціонування. На відміну від користувацьких регістрів загального призначення, системні регістри недоступні у звичайному режимі виконання програм – звернення до них можливе лише з **привілейованого режиму**, тобто з коду операційної системи або драйверів. Спроба звернутись до них з програми користувача спричиняє виключення захисту.

Регістри керування

Регістр **CR0** – найважливіший з регістрів керування. Його окремі біти вмикають або вимикають ключові режими роботи процесора:

Біт	Назва	Призначення
0	PE (Protection Enable)	вмикає захищений режим роботи процесора
1	MP (Monitor Coprocessor)	керує взаємодією з співпроцесором
2	EM (Emulation)	вмикає емуляцію співпроцесора

Біт	Назва	Призначення
16	WP (Write Protect)	забороняє запис у захищені сторінки пам'яті
31	PG (Paging)	вмикає сторінкову організацію пам'яті

Найважливіші біти – PE і PG. Саме встановлення біта PE переводить процесор із реального режиму (в якому він стартує після увімкнення) у захищений режим де стає доступна вся 32-бітна адресація і механізми захисту пам'яті. Біт PG вмикає сторінкову організацію пам'яті без якої неможлива віртуальна пам'ять.

CR1 – зарезервований, не використовується

CR2 – містить лінійну адресу, що спричинила останнє виключення сторінки (page fault)

CR3 – містить фізичну адресу каталогу сторінок поточного процесу

CR3 є ключовим для реалізації віртуальної пам'яті – при кожному перемиканні між процесами операційна система записує в CR3 адресу таблиці сторінок нового процесу, що миттєво змінює весь адресний простір.

Регістри таблиць дескрипторів

Ці регістри зберігають адреси і розміри системних таблиць, що описують сегменти пам'яті і переривання:

Регістр	Назва	Призначення
GDTR	Global Descriptor Table Register	адреса і розмір глобальної таблиці дескрипторів
LDTR	Local Descriptor Table Register	адреса і розмір локальної таблиці дескрипторів
IDTR	Interrupt Descriptor Table Register	адреса і розмір таблиці дескрипторів переривань
TR	Task Register	дескриптор поточного завдання

GDT (Global Descriptor Table) – таблиця, що описує всі сегменти пам'яті доступні в системі: їх базові адреси, розміри і права доступу. Саме через GDT реалізується захист пам'яті між процесами.

IDT (Interrupt Descriptor Table) – таблиця, що містить адреси обробників усіх переривань і виключень. Коли виникає переривання, процесор знаходить у IDT адресу відповідного обробника і передає йому управління.

Регістри налагодження

Вісім регістрів **DR0–DR7** реалізують апаратні точки зупинки (hardware breakpoints):

DR0–DR3 – адреси чотирьох точок зупинки

DR4–DR5 – зарезервовані

DR6 – регістр стану – яка точка зупинки спрацювала

DR7 – регістр керування – умови спрацювання кожної точки

Апаратні точки зупинки є потужнішими за програмні – вони можуть зупиняти виконання не лише при досягненні певної інструкції, але і при **читанні або записі** певної адреси пам'яті. Саме це використовують налагоджувачі (зокрема Visual Studio) коли встановлюється точка зупинки на зміну змінної.

Машинно-залежні регістри MSR

MSR (Model Specific Registers) – велика група регістрів специфічних для конкретної моделі процесора, що використовуються для управління процесором, контролю за його продуктивністю, отримання інформації про помилки. Ці регістри доступні через спеціальні інструкції RDMSR і WRMSR та містять:

- лічильники продуктивності (кількість виконаних інструкцій, кількість промахів кешу)
- налаштування енергоспоживання
- керування технологіями віртуалізації (Intel VT-x)
- базові адреси для інструкцій SYSCALL/SYSRET

5.3. Регістр прапорців EFLAGS

Регістр EFLAGS (рис. 5.4) – це 32-бітний регістр, кожен біт якого є окремим прапорцем, що відображає або поточний стан процесора, або результат останньої арифметичної чи логічної операції. На відміну від регістрів загального призначення, програміст не може довільно записати значення в EFLAGS через звичайну інструкцію запису MOV – прапорці встановлюються і скидаються автоматично як побічний ефект виконання інструкцій. Прапорці поділяються на три групи: прапорці стану, прапорці керування і системні прапорці.

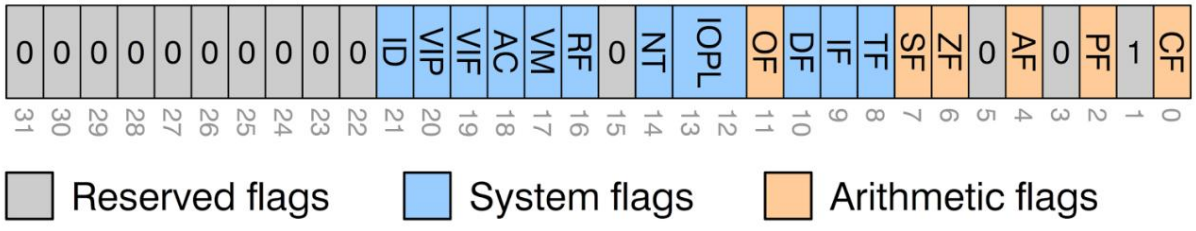

Рисунок 5.4 – Структура регістру EFLAGS [23]

Прапорці стану

Встановлюються автоматично після арифметичних і логічних операцій і використовуються командами умовного переходу. Прапорці стану є фундаментом для організації умовних переходів і циклів, які будуть розглядатись пізніше.

CF – Carry Flag (прапорець перенесення)

Встановлюється в 1 якщо результат операції додавання спричинив перенесення з старшого біта за межі розрядної сітки, або якщо при відніманні виникло позичання в старший біт. Фактично CF сигналізує про те, що результат не вміщується в розрядну сітку для беззнакових чисел – тобто є прапорцем переповнення для беззнакової арифметики. Також використовується при багаторозрядних обчисленнях де результат займає більше одного регістра – інструкція ADC (Add with Carry) враховує CF від попереднього розряду при додаванні наступного.

```
MOV EAX, 0FFFFFFFFh   ; максимальне беззнакове 32-бітне число
ADD EAX, 1            ; результат 0, CF = 1 (перенесення з 32-го біта)
```

PF – Parity Flag (прапорець парності)

Встановлюється в 1 якщо молодший байт результату містить парну кількість одиничних бітів, і скидається в 0 якщо кількість одиничних бітів непарна. Цей прапорець є пережитком ранніх комп'ютерних систем де він використовувався для перевірки цілісності даних при передачі. У сучасному програмуванні використовується вкрай рідко.

```
MOV AL, 00000011b     ; два одиничних біти – парна кількість, PF = 1
MOV AL, 00000111b     ; три одиничних біти – непарна кількість, PF = 0
```

AF – Auxiliary Carry Flag (додатковий прапорець перенесення)

Встановлюється в 1 якщо в результаті операції виникло перенесення з третього біта в четвертий – тобто перенесення всередині байта між його молодшою і старшою тетрадами. Використовується виключно для підтримки двійково-десяткової арифметики (BCD – Binary Coded Decimal) де кожна десяткова цифра кодується чотирма бітами. Інструкції DAA і DAS використовують AF для корекції результату після додавання і віднімання BCD чисел.

```
MOV AL, 09h
ADD AL, 01h     ; 9 + 1 = 10, перенесення з біта 3 в біт 4, AF = 1
```

ZF – Zero Flag (прапорець нуля)

Встановлюється в 1 якщо результат операції дорівнює нулю, і скидається в 0 в усіх інших випадках. Є найчастіше використовуваним прапорцем – на ньому базується більшість умовних переходів. Важливо розуміти, що ZF встановлюється не лише арифметичними операціями, але й логічними – AND, OR, XOR також впливають на ZF. Команда CMP яка виконує віднімання без збереження результату використовується саме для встановлення ZF і інших прапорців перед командою умовного переходу.

```
SUB EAX, EAX     ; EAX = 0, ZF = 1
CMP EAX, EBX     ; якщо EAX = EBX то різниця 0 і ZF = 1
```

SF – Sign Flag (прапорець знаку)

Завжди дорівнює старшому біту результату операції – тобто біту, що є знаковим у доповняльному коді. Якщо результат від'ємний, старший біт дорівнює 1 і SF = 1. Якщо результат додатний або нульовий – SF = 0. Прапорець має сенс лише при знаковій арифметиці – для беззнакових чисел старший біт є просто старшим бітом числа, а не знаком.

```
MOV EAX, 5
SUB EAX, 10      ; результат -5, старший біт 1, SF = 1
```

OF – Overflow Flag (прапорець переповнення)

Встановлюється в 1 якщо результат знакової операції вийшов за межі допустимого діапазону – тобто коли математично правильний результат не може бути коректно представлений у відведеній розрядній сітці зі знаком. Це відбувається у двох випадках: коли сума двох додатних чисел дає від'ємний результат, або коли сума двох від'ємних чисел дає додатній результат. OF є прапорцем переповнення для знакової арифметики – на відміну від CF який виконує цю роль для беззнакової. Саме тому процесор встановлює обидва прапорці незалежно – програміст сам вирішує які з них перевіряти залежно від того чи є числа знаковими.

```
MOV EAX, 7FFFFFFFh    ; максимальне додатне 32-бітне знакове число
ADD EAX, 1            ; результат 80000000h = від'ємне число, OF = 1
```

Прапорець керування

DF – Direction Flag (прапорець напрямку)

На відміну від прапорців стану, DF не встановлюється автоматично – він керує поведінкою рядкових операцій (MOVS, CMPS, SCAS, LODS, STOS) і

змінюється лише явними інструкціями. При DF = 0 рядкові операції обробляють дані у напрямку зростання адрес – від меншої адреси до більшої, тобто зліва направо. При DF = 1 напрямок змінюється на протилежний – від більшої адреси до меншої. Для встановлення DF використовується інструкція STD, для скидання – CLD. У більшості випадків рекомендується явно скидати DF перед рядковими операціями щоб гарантувати правильний напрямок обробки незалежно від попереднього стану.

```
CLD     ; DF = 0, обробка зліва направо
STD     ; DF = 1, обробка справа наліво
```

Системні прапорці

До групи системних прапорців входять 8 системних прапорців, які керують введенням-виведенням, маскованим перериванням, відлагоджуванням, перемиканням між задачами та режимом віртуального процесора i8086. Прикладним програмам не рекомендується модифікувати без необхідності ці прапорці, через те, що у більшості випадків це спричиняє зупинку роботи програми. Системні прапорці використовуються операційною системою або драйверами і, як правило, недоступні для зміни з програм користувача:

TF – Trap Flag (прапорець пастки)

При TF = 1 процесор переходить у покроковий режим виконання – після кожної інструкції генерується відладочне переривання. Саме цей механізм використовує налагоджувач Visual Studio при покроковому виконанні програми. Програміст безпосередньо не керує TF – це робить налагоджувач.

IF – Interrupt Flag (прапорець переривань)

Керує реакцією процесора на апаратні переривання від зовнішніх пристроїв. При IF = 1 процесор реагує на переривання і передає керування відповідному обробнику. При IF = 0 апаратні переривання ігноруються – це використовується в критичних секціях коду операційної системи де переривання не допускаються. Встановлюється інструкцією STI, скидається інструкцією CLI – обидві доступні лише з привілейованого режиму.

IOPL – I/O Privilege Level (рівень привілеїв введення-виведення)

Два біти, що визначають мінімальний рівень привілеїв необхідний для виконання інструкцій введення-виведення (IN, OUT). У звичайних програмах користувача прямий доступ до портів введення-виведення заборонений операційною системою.

NT – Nested Task

Встановлюється коли поточне завдання було викликане через інструкцію CALL, а не через звичайний перехід. Використовується механізмом перемикання завдань процесора для правильного повернення до попереднього завдання через інструкцію IRET.

RF – Resume Flag

При RF = 1 забороняє повторне спрацювання апаратної точки зупинки після відновлення виконання. Автоматично скидається після виконання першої інструкції – це дозволяє налагоджувачу відновити виконання програми не потрапляючи в нескінченний цикл обробки одного і того ж переривання.

VM – Virtual-8086 Mode

При VM = 1 процесор переходить у режим Virtual-8086, що дозволяє виконувати 16-бітні програми написані для реального режиму в середовищі захищеного режиму. Використовувався у Windows 9x для запуску програм DOS.

AC – Alignment Check

При AC = 1 процесор генерує виключення якщо програма звертається до пам'яті за невирівняною адресою – наприклад читає 4-байтне значення з адреси, що не кратна 4. Використовується для налагодження програм де невирівняний доступ до пам'яті є помилкою.

VIF – Virtual Interrupt Flag

Віртуальний аналог прапорця IF для режиму Virtual-8086. Дозволяє операційній системі емулювати поведінку IF для гостьових програм не надаючи їм реального доступу до керування перериваннями.

VIP – Virtual Interrupt Pending

Встановлюється в 1 щоб сигналізувати операційній системі про наявність відкладеного віртуального переривання яке очікує обробки. Використовується спільно з VIF у режимі Virtual-8086.

ID – ID Flag

Якщо операційна система може встановити і скинути цей біт – процесор підтримує інструкцію CPUID. Інструкція CPUID дозволяє програмно отримати інформацію про процесор: виробника, модель, підтримувані набори інструкцій (MMX, SSE тощо).

Варто звернути увагу, що безпосереднього читання та запису регістру EFLAGS через MOV немає. Для роботи з регістром використовуються спеціальні інструкції, що передають його вміст через стек:

```
PUSHFD      ; зберегти EFLAGS на стек (32-бітна версія)
POPFD       ; відновити EFLAGS зі стека
LAHF        ; завантажити молодший байт EFLAGS в регістр AH
SAHF        ; зберегти AH в молодший байт EFLAGS
```

Типовий приклад збереження і відновлення прапорців, що використовується коли необхідно виконати операції не змінюючи поточного стану прапорців:

```
PUSHFD      ; зберігаємо поточний стан EFLAGS
. . .       ; якийсь код, що змінює прапорці
POPFD       ; відновлюємо попередній стан EFLAGS
```

5.4. Організація та принцип роботи зі стеком

Стек – це область оперативної пам'яті з особливим способом організації доступу до даних. На відміну від довільного доступу до пам'яті де можна прочитати або записати будь-яку комірку за її адресою, стек працює за принципом **LIFO** (Last In – First Out, останній прийшов – перший вийшов): елемент, що був доданий останнім буде вилучений першим. Класична аналогія – стопка тарілок, де можна покласти тарілку лише зверху і взяти лише верхню. Стек в архітектурі IA-32 має важливу особливість – він **росте у бік зменшення адрес** (рис. 5.5). Тобто, коли до стека додається новий елемент, адреса вершини стека зменшується, а не збільшується як можна було б очікувати інтуїтивно.

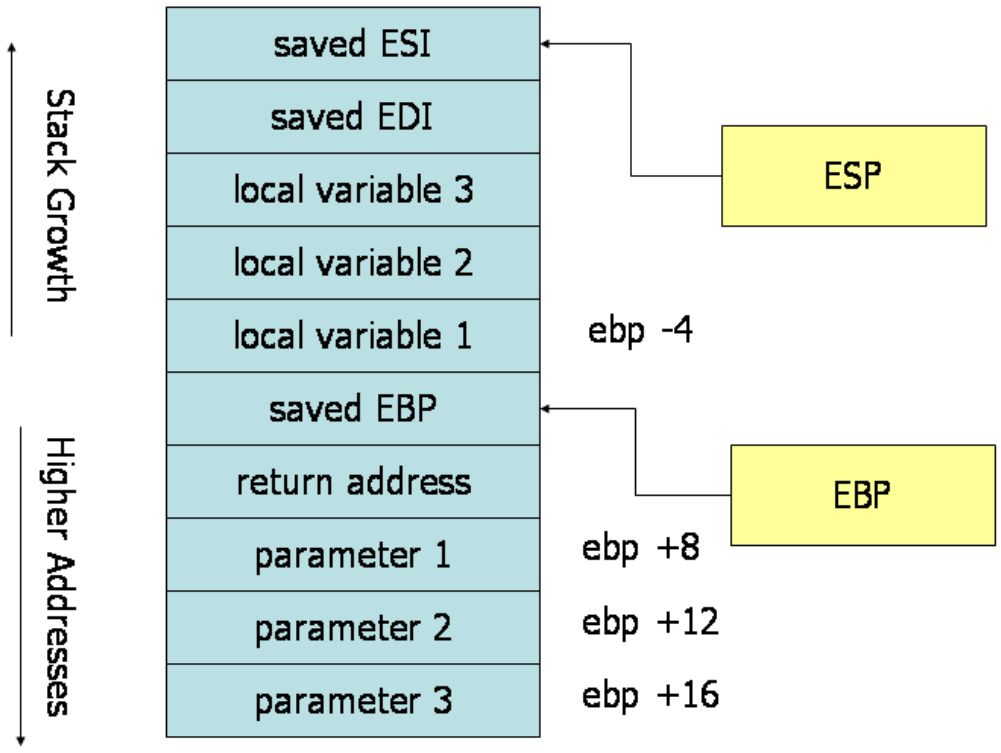

Рисунок 5.5 – Організація стеку [22]

Поточна адреса вершини стека завжди зберігається в регістрі **ESP** (Extended Stack Pointer). При додаванні елемента ESP зменшується на розмір елемента, при вилученні – збільшується.

Основні інструкції роботи зі стеком:

PUSH – додає значення на вершину стека. Виконує дві дії: зменшує ESP на 4 (розмір 32-бітного значення) і записує значення за новою адресою ESP:

```
PUSH EAX     ; ESP = ESP - 4, [ESP] = EAX
PUSH 42      ; ESP = ESP - 4, [ESP] = 42
```

POP – вилучає значення з вершини стека. Читає значення за поточною адресою ESP і збільшує ESP на 4:

```
POP EBX     ; EBX = [ESP], ESP = ESP + 4
```

Важливо розуміти, що POP не стирає значення з пам'яті – воно залишається там до наступного запису. Просто ESP переміщується і це місце вважається вільним.

Збереження і відновлення регістрів

Одне з найважливіших застосувань стека – тимчасове збереження регістрів. Оскільки регістрів загального призначення лише вісім, виникають ситуації коли потрібний регістр вже використовується і його значення не можна втрачати. Стек вирішує цю проблему:

```
PUSH EAX          ; зберігаємо EAX на стек
PUSH EBX          ; зберігаємо EBX на стек

...               ; використовуємо регістри EAX і EBX для інших цілей

POP  EBX              ;  відновлюємо значення EBX (порядок зворотний
збереженню!)
POP EAX           ; відновлюємо EAX
```

Порядок відновлення обов'язково має бути зворотним до порядку збереження – саме через принцип LIFO. Це типова помилка початківців.

Для збереження і відновлення всіх регістрів загального призначення (AX, CX, DX, BX, SP, BP, SI, DI) одночасно існують спеціальні інструкції:

```
PUSHA     ; зберегти всі регістри загального призначення на стек
POPA      ; відновити всі регістри загального призначення зі стека
```

Для збереження і відновлення 32 бітних регістрів виконується відповідно команди PUSHAD та POPAD. Ці команди часто використовується в обробниках переривань або перед викликом функцій, щоб гарантувати збереження стану регістрів

Стек і асемблерні вставки у Visual Studio

При використанні блоку __asm {} особливо важливо дотримуватись правила збереження і відновлення регістрів. Компілятор C очікує, що після виконання блоку __asm значення регістрів ESP і EBP залишаться незмінними – порушення цього правила призведе до непередбачуваної поведінки програми або її аварійного завершення. Регістри EAX, EBX, ECX, EDX змінювати можна, але якщо їх значення потрібні навколишньому коду C – їх слід зберегти перед блоком і відновити після:

```
int a = 10, result = 0;

__asm {
    PUSH EBX          ; зберігаємо EBX бо будемо його змінювати
    MOV EAX, a
    MOV EBX, 3
    IMUL EBX          ; EAX = EAX * EBX
    MOV result, EAX
    POP EBX           ; відновлюємо EBX
}
```

Стек і виклик функцій

Стек відіграє ключову роль при виклику функцій – через нього передаються аргументи функції і зберігається адреса повернення. Коли виконується інструкція CALL, адреса наступної інструкції автоматично зберігається на стек, а інструкція RET знімає цю адресу зі стека і передає туди керування. Детально цей механізм буде розглянуто у підрозділі про організацію переходів і виклик функцій.

Переповнення стека

Стек має обмежений розмір, що визначається операційною системою – як правило 1–8 МБ для програм користувача. Якщо програма додає на стек більше даних ніж відведено, виникає **переповнення стека** (stack overflow) – запис виходить за межі відведеної області і перезаписує сусідні ділянки пам'яті. Це призводить до аварійного завершення програми або – у гіршому випадку – до вразливості безпеки, що може бути використана зловмисником. Рекурсивні функції без умови завершення є класичним прикладом, що призводить до переповнення стека.

5.5. Синтаксис мови асемблера

Синтаксис мови асемблера значно простіший за синтаксис мов високого рівня – немає складних конструкцій, типів даних, об'єктів чи функцій у звичному розумінні. Програма складається з послідовності інструкцій, кожна з яких записується в окремому рядку і має чітку структуру:

```
мітка: мнемоніка операнд_призначення, операнд_джерела  ;
                                коментар
```

Єдиним обов'язковим елементом є **мнемоніка** – решта є необов'язковими залежно від конкретної інструкції. Послідовно розглянемо кожен з елементів.

Мітки.

Мітка – це символічне ім'я, що позначає адресу поточної інструкції в пам'яті. Завершується двокрапкою і може містити літери, цифри і символ підкреслення, але не може починатись з цифри:

```
loop_start:    MOV ECX, 10
cycle:         ADD EAX, EBX
```

Мітки використовуються як цілі для команд переходу і циклів. Асемблер під час трансляції замінює символічні імена міток на реальні числові адреси.

Мнемоніки.

Мнемоніка – коротке текстове позначення машинної інструкції. Визначає яку саме операцію виконає процесор. У синтаксисі Visual Studio мнемоніки не чутливі до регістру – MOV, mov і Mov є еквівалентними записами, хоча прийнято писати великими літерами:

```
MOV     ; переміщення даних
ADD     ; додавання
SUB     ; віднімання
MUL     ; множення беззнакове
IMUL    ; множення знакове
DIV     ; ділення беззнакове
IDIV    ; ділення знакове
CMP     ; порівняння
JMP     ; безумовний перехід
```

Операнди.

Більшість інструкцій мають два операнди. У синтаксисі Intel, що використовується у Visual Studio діє стандартне правило подання операндів:

```
інструкція операнд-призначення, операнд-джерело
```

Перший операнд завжди є операндом призначення – куди записується результат. Другий – джерелом звідки береться значення. Це важливо запам'ятати, оскільки існує також синтаксис AT&T [19] де порядок операндів протилежний і плутанина між ними є досить поширеною помилкою [20].

> NB! Для більшості команд асемблера обидва операнди повинні мати однакову розрядність – 8, 16 або 32 біти. Спроба виконати операцію між операндами різної розрядності є помилкою і буде відхилена асемблером на етапі трансляції. Винятком є команди, що явно призначені для зміни розрядності: MOVSX, MOVZX, CBW, CWD, CWDE, CDQ – вони за визначенням працюють з операндами різного розміру.

Операнди бувають трьох видів:

Регістр – один з регістрів процесора розглянутих у розділі 5.2:

```
MOV EAX, EBX     ; запис значення з регістра EBX в регістр EAX
```

Безпосереднє значення – числова константа вбудована безпосередньо в інструкцію. Може записуватись у різних системах числення:

```
MOV EAX, 42          ; десяткове
MOV EAX, 2Ah         ; шістнадцяткове (суфікс h)
MOV EAX, 00101010b   ; двійкове (суфікс b)
```

Адреса пам'яті – звернення до комірки пам'яті через квадратні дужки. Значення у дужках є адресою комірки, а не самим значенням:

```
MOV EAX, [1000h]       ; читаємо значення з адреси 1000h
MOV [1000h], EAX       ; записуємо EAX за адресою 1000h
MOV EAX, [EBX]         ; читаємо значення з адреси, що зберігається в
EBX
MOV EAX, [EBX + 4]     ; читаємо значення з адреси EBX + 4
```

Змінна мови С

У блоці __asm {} Visual Studio дозволяє використовувати імена змінних оголошених у коді мови С безпосередньо як операнди – асемблер автоматично замінює ім'я змінної на її адресу в пам'яті:

```
asm
int a = 10, b = 5, result = 0;

__asm {
    MOV EAX, a          ; завантажити значення змінної a в EAX
    ADD EAX, b          ; додати значення змінної b
    MOV result, EAX     ; зберегти результат у змінну result
}
```

Це є суттєвою перевагою асемблерних вставок порівняно з окремими асемблерними файлами – не потрібно вручну обчислювати адреси змінних або передавати їх через регістри. Асемблер самостійно визначає де в пам'яті розташована змінна і генерує відповідну адресацію.

Важливо розуміти, що звернення до змінної в асемблері – це завжди звернення до пам'яті, а не до регістра. Тому пряма операція між двома змінними неможлива – в архітектурі IA-32 більшість інструкцій не дозволяють мати обидва операнди в пам'яті одночасно:

```
asm
MOV EAX, a      ; правильно – один операнд пам'ять, другий регістр
MOV b, EAX      ; правильно – один операнд пам'ять, другий регістр
MOV a, b        ; ПОМИЛКА – обидва операнди в пам'яті
```

Саме тому правильною практикою роботи зі змінними в асемблері передбачає завантаження змінної в регістр, виконання операції і збереження результату назад у змінну.

Режими адресації

Режим адресації визначає спосіб обчислення адреси операнда. Архітектура IA-32 підтримує кілька режимів:

Регістрова адресація – операнд знаходиться безпосередньо в регістрі:

```
MOV EAX, EBX    ; операнд – регістр EBX
```

Безпосередня адресація – операнд є константою вбудованою в інструкцію:

```
MOV EAX, 10     ; операнд – константа 10
```

Пряма адресація – операнд знаходиться в пам'яті за фіксованою адресою:

```
MOV EAX, [1000h]     ; операнд знаходиться за адресою 1000h
```

Непряма регістрова адресація – адреса операнда зберігається в регістрі:

```
MOV EAX, [EBX]     ; адреса операнда зберігається в EBX
```

Адресація зі зміщенням – адреса обчислюється як сума регістра і константи. Використовується для доступу до елементів масивів і полів структур:

```
MOV EAX, [EBX + 8]     ; адреса = EBX + 8
```

Адресація з індексом і масштабом – адреса обчислюється за формулою: база + індекс × масштаб + зміщення. Масштаб може бути 1, 2, 4 або 8, що відповідає розмірам типів даних byte, word, dword і qword:

```
MOV EAX, [EBX + ECX*4 + 8]     ; адреса = EBX + ECX×4 + 8
```

Цей режим є особливо зручним для роботи з масивами – EBX містить базову адресу масиву, ECX є індексом елемента, 4 є розміром одного елемента в байтах (для 32-бітних цілих), а 8 є зміщенням до початку масиву всередині структури.

Розміри операндів

Ще одна з особливостей асемблера – необхідність явно вказувати розмір операнда у деяких випадках. Коли асемблер не може визначити розмір операнда самостійно – наприклад при записі константи за адресою пам'яті – використовуються спеціальні префікси:

```
MOV BYTE PTR [EBX], 10     ; записати 1 байт
MOV WORD PTR [EBX], 10     ; записати 2 байти
MOV DWORD PTR [EBX], 10    ; записати 4 байти
```

Проте, якщо один з операндів є регістром, розмір визначається автоматично: AL відповідає 1 байту, AX – 2 байтам, EAX – 4 байтам.

Приховані операнди

Деякі інструкції неявно використовують фіксовані регістри, які не записуються в інструкції, але обов'язково задіяні. Які конкретно неявні операнди будуть задіяні асемблер визначає за розміром явного операнда. Тим не менш, знання прихованих операндів є критично важливим – їх використання може несподівано змінити значення регістрів:

Інструкція	Явний операнд	Приховані операнди
MUL x	int x	множене: EAX; результат: EDX:EAX
DIV x	short int x	ділене: DX:AX; частка: AX; остача: DX
PUSH x	x	неявно змінює ESP
POP y	y	неявно змінює ESP
LOOP мітка	мітка	неявно використовує і зменшує ECX

Варто звернути увагу на певну асиметрію – при множенні неявний операнд (множник) відповідає за розміром явному операнду і знаходиться в молодшому регістрі-акумуляторі (AL, AX, EAX), а результат множення (добуток) завжди вдвічі ширший за явний операнд (AX, DX:AX, EDX:EAX), а при діленні навпаки неявний операнд (ділене) є вдвічі ширший за явний (дільник) і оскільки ділення є оберненою операцією множення – результат зберігається в регістрі-акумуляторі. Наприклад інструкція MUL EBX виконує множення EAX на EBX і записує 64-бітний результат у пару регістрів EDX:EAX – старші 32 біти в EDX, молодші в EAX. Якщо програміст забуде про це і використовує EDX для інших цілей, його значення буде втрачено.

Коментарі

У блоці __asm {} у Visual Studio підтримуються два стилі коментарів:

- крапка з комою – відділяє код команди від коментаря в стилі асемблера.

```
MOV EAX, 0      ; коментар у стилі асемблера – до кінця рядка
```

- подвійний слеш та слеші з зірочками – коментарі відділяються в стилі мови С.

```
MOV EAX, 0      // коментар у стилі C++ – до кінця рядка
/* багаторядковий коментар у стилі C */
```

Коментарі є особливо важливими в асемблері – на відміну від мов високого рівня де намір програміста часто зрозумілий з коду, асемблерні інструкції самі по собі не описують, що саме обчислюється. Рекомендується коментувати не, що робить інструкція (це очевидно з мнемоніки), а навіщо вона тут знаходиться.

5.6. Основні команди для роботи з даними

Основні команди можна розділити на декілька основних груп: команди пересилання даних, арифметичні команди та логічні команди. Проте їх функціонал може бути схожим.

Команди пересилання даних:

MOV – команда пересилання даних.

Це одна з найчастіше використовуваних команд асемблера. Копіює значення з операнда-джерела в операнд-призначення не змінюючи значення джерела і не встановлюючи жодних прапорців EFLAGS.

```
MOV операнд-призначення, операнд-джерело
```

Допустимі комбінації операндів:

Операнд призначення	Операнд-джерело
регістр	регістр
регістр	пам'ять
регістр	константа
пам'ять	регістр
пам'ять	константа

Комбінація пам'ять-пам'ять є недопустимою – як зазначалось у попередньому розділі, для копіювання між двома комірками пам'яті необхідно використовувати проміжний регістр.

Також логічно, що операндом призначення не можуть бути безпосередні дані, оскільки для їх зберігання не виділяється пам’ять.

Важливим моментом є те, що розміри операндів призначення і джерела мають обов'язково збігатись:

```
MOV EAX, EBX        ; 32 біт → 32 біт
MOV AX, BX          ; 16 біт → 16 біт
MOV AL, BL          ; 8 біт → 8 біт
MOV EAX, AX         ; ПОМИЛКА – розміри не збігаються
```

XCHG – обмін значеннями.

Обмінює значення двох операндів місцями. На відміну від MOV виконує двосторонню передачу даних за одну інструкцію без використання проміжного регістра:

```
XCHG операнд1, операнд2
XCHG EAX, EBX     ; EAX ←→ EBX
XCHG EAX, [EBX]   ; EAX ←→ значення в пам'яті за адресою, що міститься в EBX
```

Комбінація пам'ять-пам'ять та використання як операнда безпосередніх даних також є недопустимими. Варто зазначити, що XCHG з операндом у пам'яті автоматично використовує механізм блокування шини пам'яті, що робить операцію атомарною (виконується як єдине, неподільне ціле, тобто гарантує, що під час її роботи ніщо не може перервати процес або змінити дані, з якими вона працює) – це використовується в багатопоточному програмуванні для синхронізації.

MOVSX – пересилання з розширенням знаку.

Копіює значення меншого розміру у регістр більшого розміру з розширенням знаку – старші біти заповнюються значенням знакового біта джерела. Операндом-джерелом може бути регістр, значення в пам’яті та безпосередні дані. Використовується для знакових цілих чисел (signed):

```
MOVSX призначення, джерело
MOVSX EAX, BX       ; розширення 16-бітного BX до 32-бітного EAX зі знаком
MOVSX EAX, BL       ; розширення 8-бітного BL до 32-бітного EAX зі знаком
MOVSX AX, BL        ; розширення 8-бітного BL до 16-бітного AX зі знаком
```

Приклад розширення числа −5:

```
BL  =  11111011    (−5 у доповняльному коді, 8 біт)
EAX =  11111111 11111111 11111111 11111011    (−5 у 32 бітах)
```

Знаковий біт 1 заповнює всі старші розряди – число залишається від'ємним і математично рівним −5.

MOVZX – пересилання з розширенням нулями.

Копіює значення меншого розміру у регістр більшого розміру заповнюючи старші біти нулями незалежно від знаку. Використовується для беззнакових цілих чисел (unsigned):

```
MOVZX призначення, джерело
MOVZX EAX, BX        ; розширення 16-бітного BX до 32-бітного EAX з нулями
MOVZX EAX, BL      ; розширення 8-бітного BL до 32-бітного EAX з нулями
```

Приклад розширення числа 251 (те саме бітове представлення, що і −5, але беззнакове):

```
BL  =  11111011    (251 як беззнакове, 8 біт)
EAX =  00000000 00000000 00000000 11111011    (251 у 32 бітах)
```

Різниця між MOVSX і MOVZX принципова – для одного і того ж бітового шаблону вони дають різні результати залежно від того чи використовується число як знакове, чи ні.

CBW, CWD, CWDE, CDQ – розширення знакового цілого

На відміну від MOVSX, що вимагає явного зазначення джерела і призначення, команди CBW, CWD і CDQ працюють виключно з фіксованими прихованими регістрами і не мають жодних явних операндів:

CBW (Convert Byte to Word) – розширює знаковий байт AL до слова AX:

```
MOV AL, -5      ; попередні дії, AL = 11111011 (−5)
CBW     ; виконуємо розширення AL → AX, AX = 11111111 11111011 (−5 у 16 бітах)
```

CWD (Convert Word to Doubleword) – розширює знакове слово AX до подвійного слова DX:AX:

```
MOV AX, -5      ; AX = 1111111111111011 (−5)
CWD     ; DX:AX = 11111111111111111111111111111011 (−5 у 32 бітах)
```

CWDE (Convert Word to Doubleword Extended) – розширює знакове слово AX до подвійного слова **EAX**. Відрізняється від CWD тим, що результат записується в EAX, а не в пару DX:AX:

```
MOV AX, -5      ; AX = 1111111111111011 (−5)
CWDE     ; EAX = 11111111111111111111111111111011 (−5 у 32 бітах)
```

CDQ (Convert Doubleword to Quadword) – розширює знакове подвійне слово EAX до учетвереного слова EDX:EAX:

```
MOV EAX, -5      ; EAX = 11111111111111111111111111111011 (−5)
CDQ     ; EDX:EAX = −5 у 64 бітах
```

Усі три команди виконують те саме, що і MOVSX – розширення знаку, але без можливості вибору регістрів. Практично важливе застосування – підготовка до знакового ділення IDIV, яке вимагає щоб ділене займало пару регістрів DX:AX або EDX:EAX. Якщо ділене знаходиться лише в AX або EAX, перед IDIV необхідно виконати CWD або CDQ відповідно:

```
MOV EAX, -17     ; ділене
CDQ              ; розширюємо EAX до EDX:EAX
MOV EBX, 5      ; дільник
IDIV EBX         ; EAX = -3 (частка), EDX = -2 (остача)
```

Без CDQ значення EDX було б невизначеним, що призвело б до неправильного результату або виключення процесора.

BSWAP – зміна порядку байтів

Перед розглядом команди необхідно пояснити поняття порядку байтів. Коли багатобайтове число зберігається в пам'яті, виникає питання – в якому порядку розміщувати його байти? Існують два підходи:

Little-endian – молодший байт зберігається за меншою адресою. Саме цей порядок використовується в архітектурі IA-32:

Число 0x12345678 у пам'яті:

Адреса+0: 78 (молодший байт)

Адреса+1: 56

Адреса+2: 34

Адреса+3: 12 (старший байт)

Big-endian – старший байт зберігається за меншою адресою. Використовується в мережевих протоколах (TCP/IP) та деяких процесорних архітектурах:

Число 0x12345678 у пам'яті:

Адреса+0: 12 (старший байт)
Адреса+1: 34
Адреса+2: 56
Адреса+3: 78 (молодший байт)

При обміні даними між системами з різним порядком байтів необхідне перетворення. Саме для цього і призначена команда BSWAP:

```
BSWAP регістр
```

BSWAP міняє місцями байти 32-бітного регістра – перший з четвертим і другий з третім:

```
    ; EAX = 0x12345678
BSWAP EAX
    ; EAX = 0x78563412
```

Типове застосування – отримання даних з мережі де числа передаються у форматі big-endian і перетворення їх у little-endian для подальшої обробки:

```
unsigned int networkValue = 0x12345678; // отримано з мережі
unsigned int hostValue;

__asm {
    MOV EAX, networkValue
    BSWAP EAX
    MOV hostValue, EAX
}
```

LEA – завантаження ефективної адреси

На відміну від MOV з адресацією через дужки, що читає **значення** за адресою, LEA завантажує саму **адресу** у регістр без звернення до пам'яті:

```
LEA призначення, джерело
LEA EAX, [EBX]          ; EAX = EBX (адреса, не значення)
LEA EAX, [EBX + 8]      ; EAX = EBX + 8
LEA EAX, [EBX + ECX*4 + 8]   ; EAX = EBX + ECX×4 + 8
```

Порівняння з MOV:

```
MOV EAX, [EBX + 8]    ; EAX = значення в пам'яті за адресою (EBX + 8)
LEA EAX, [EBX + 8]    ; EAX = EBX + 8  (сама адреса)
```

LEA часто використовується не лише для обчислення адрес, але і як швидкий спосіб виконати арифметику: обчислення EBX + ECX×4 + 8 через одну інструкцію LEA швидше ніж послідовність ADD і MUL. Компілятори C активно використовують LEA саме для арифметичних обчислень.

Також LEA є зручним способом отримати адресу локальної змінної C у блоці __asm:

```
int a = 10;
int *ptr;

__asm {
    LEA EAX, a        ; EAX = адреса змінної a
    MOV ptr, EAX      ; зберігаємо адресу в ptr
}
```

LODS – завантаження рядкового елемента

Команди групи LODS належать до групи **рядкових інструкцій** (string instructions), що призначені для обробки послідовностей даних у пам'яті. LODS завантажує елемент з адреси ESI в акумулятор і автоматично змінює ESI для переходу до наступного елемента:

```
LODSB    ; завантажити байт з [ESI] в AL,  ESI = ESI ± 1
LODSW    ; завантажити слово з [ESI] в AX,  ESI = ESI ± 2
LODSD    ; завантажити подвійне слово з [ESI] в EAX, ESI = ESI ± 4
```

Напрямок зміни ESI визначається прапорцем DF – при DF = 0 адреса збільшується (рух вперед), при DF = 1 – зменшується (рух назад).

LODS завжди має приховані операнди: операнд-джерело завжди [ESI], операнд-призначення завжди AL/AX/EAX залежно від варіанту команди.

Арифметичні команди

ADD – додавання

Додає значення операнду-джерела до операнда-призначення і записує результат у операнд-призначення. Встановлює прапорці CF, ZF, SF, OF, PF, AF:

```
ADD призначення, джерело
ADD EAX, EBX        ; EAX = EAX + EBX
ADD EAX, 10         ; EAX = EAX + 10
ADD EAX, a          ; EAX = EAX + значення змінної a
ADD [EBX], EAX      ; значення в пам'яті = значення в пам'яті + EAX
```

ADC – додавання з перенесенням

Виконує те саме, що й ADD, але додатково додає значення прапорця CF. Використовується для додавання чисел, що не вміщуються в один регістр – так звана багаторозрядна арифметика:

```
ADC призначення, джерело
```

Приклад додавання двох 64-бітних чисел на 32-бітному процесорі. Нехай перше число зберігається в парі EDX:EAX, друге в парі ECX:EBX:

```
ADD EAX, EBX      ; додаємо молодші 32 біти, CF = перенесення
ADC EDX, ECX      ; додаємо старші 32 біти враховуючи CF
```

Без ADC перенесення з молодших розрядів у старші було б втрачено.

SUB – віднімання

Віднімає значення операнда-джерела від операнда-призначення і записує результат у операнд-призначення. Встановлює ті самі прапорці, що й ADD:

```
SUB призначення, джерело
SUB EAX, EBX        ; EAX = EAX - EBX
SUB EAX, 10         ; EAX = EAX - 10
SUB EAX, a          ; EAX = EAX - значення змінної a
```

SBB – віднімання з позичанням

Аналог ADC для віднімання – виконує SUB і додатково віднімає значення прапорця CF. Використовується для багаторозрядного віднімання:

```
SBB призначення, джерело
```

Приклад віднімання двох 64-бітних чисел:

```
SUB EAX, EBX      ; віднімаємо молодші 32 біти, CF = позичання
SBB EDX, ECX      ; віднімаємо старші 32 біти враховуючи CF
```

MUL – беззнакове множення

```
MUL джерело
```

Множить явний операнд на прихований регістр-акумулятор. Розмір прихованих операндів визначається розміром явного:

Розмір джерела	Множене	Результат
8 біт	AL	AX
16 біт	AX	DX:AX
32 біт	EAX	EDX:EAX

```
MOV AL, 25
MOV BL, 10
MUL BL          ; AX = AL × BL = 250
MOV EAX, 100000
MOV EBX, 100000
MUL EBX         ; EDX:EAX = EAX × EBX = 10 000 000 000
```

Якщо результат вміщується в молодшу половину (AX, AX або EAX відповідно) – CF і OF скидаються в 0. Якщо результат займає всю пару регістрів – CF і OF встановлюються в 1, що сигналізує про те, що старша половина результату є ненульовою.

IMUL – знакове множення

Виконує знакове множення. Має три форми запису на відміну від MUL:

Одноопера́ндна форма – повністю аналогічна MUL, але для знакових чисел:

```
IMUL EBX        ; EDX:EAX = EAX × EBX (знакове)
```

Двоопера́ндна форма – результат записується в перший операнд, джерелом є другий. Результат обрізається до розміру операнда – старша половина не зберігається:

```
IMUL EAX, EBX       ; EAX = EAX × EBX
IMUL EAX, 10        ; EAX = EAX × 10
```

Трьохоперандна форма – перший операнд є призначенням, другий джерелом, третій константою-множником:

```
IMUL EAX, EBX, 10  ; EAX = EBX × 10
```

Двох і трьохоперандні форми зручніші для використання у більшості випадків оскільки не змінюють EDX і не потребують попереднього завантаження значення в EAX.

DIV – беззнакове ділення

Ділить прихований регістр-акумулятор на явний операнд. Розмір прихованих операндів визначається розміром явного:

```
DIV дільник
```

Розмір дільника	Ділене	Частка	Остача
8 біт	AX	AL	AH
16 біт	DX:AX	AX	DX
32 біт	EDX:EAX	EAX	EDX

```
MOV AX, 100      ; ділене
MOV BL, 7        ; дільник
DIV BL           ; AL = 14 (частка), AH = 2 (остача)
```

Важливо: якщо частка не вміщується у призначений регістр або дільник дорівнює нулю – процесор генерує виключення ділення на нуль (Divide Error, переривання INT 0). Для 32-бітного ділення перед DIV необхідно обнулити EDX якщо ділене є 32-бітним числом, а не 64-бітним:

```
MOV EAX, 1000000
MOV EDX, 0         ; обнуляємо старшу частину діленого
MOV EBX, 7
DIV EBX            ; EAX = 142857 (частка), EDX = 1 (остача)
```

IDIV – знакове ділення

Повністю аналогічне DIV, але для знакових чисел. Перед IDIV необхідно розширити знак діленого за допомогою CDQ (для 32-бітного) або CWD (для 16-бітного):

```
MOV EAX, -17
CDQ                ; розширюємо знак EAX в EDX:EAX
MOV EBX, 5
IDIV EBX           ; EAX = -3 (частка), EDX = -2 (остача)
```

Знак остачі при IDIV завжди збігається зі знаком діленого – це стандартна поведінка визначена архітектурою.

INC – інкремент

Збільшує значення операнда на 1. Встановлює ZF, SF, OF, PF, AF, але **не змінює CF** – це важлива відмінність від ADD з константою 1:

```
INC операнд
INC EAX              ; EAX = EAX + 1
INC BYTE PTR [EBX] ; збільшити байт у пам'яті на 1
```

DEC – декремент

Зменшує значення операнда на 1. Так само як INC не змінює CF:

```
DEC операнд
DEC EAX               ; EAX = EAX - 1
DEC DWORD PTR [EBX] ; зменшити подвійне слово у пам'яті на 1
```

Незмінність CF при INC і DEC є принциповою – це дозволяє використовувати їх як лічильники всередині багаторозрядних арифметичних операцій де CF несе інформацію про перенесення між розрядами і не повинен бути змінений лічильником циклу.

NEG – зміна знаку

Замінює значення операнда його доповняльним кодом – тобто виконує арифметичне заперечення. Еквівалентно відніманню операнда від нуля:

```
NEG операнд
MOV EAX, 5
```

```
NEG EAX          ; EAX = -5
MOV EAX, -3
NEG EAX          ; EAX = 3
```

Встановлює CF в 0 якщо операнд був нульовим, і в 1 в усіх інших випадках. Встановлює OF якщо операнд був мінімальним від'ємним числом (80000000h для 32 біт) – в цьому випадку результат не змінюється через переповнення і OF сигналізує про помилку.

Типове застосування – обчислення абсолютного значення:

```
    MOV EAX, a
    CMP EAX, 0       ; порівнюємо з нулем
    JGE positive     ; якщо EAX ≥ 0 – пропускаємо NEG
    NEG EAX          ; інакше змінюємо знак
positive:
    MOV result, EAX
```

CMP – порівняння

Виконує віднімання джерела від призначення, але **не зберігає результат** – лише встановлює прапорці. Фактично це команда SUB без запису результату:

```
CMP операнд1, операнд2
CMP EAX, EBX       ; встановлює прапорці як EAX - EBX, але EAX не змінюється
CMP EAX, 0       ; перевіряємо чи EAX дорівнює нулю
CMP EAX, a       ; порівнюємо EAX зі змінною a
```

Прапорці після CMP інтерпретуються залежно від того чи є числа знаковими:

Умова	Беззнакові (CF, ZF)	Знакові (SF, OF, ZF)
операнд1 = операнд2	ZF=1	ZF=1
операнд1 > операнд2	CF=0, ZF=0	SF=OF, ZF=0
операнд1 < операнд2	CF=1	SF≠OF

CMP майже завжди використовується безпосередньо перед командою умовного переходу – детально це розглядається у розділі про організацію переходів.

Логічні команди

AND – логічне І

Виконує побітову кон'юнкцію операндів і записує результат у призначення. Скидає CF і OF в 0, встановлює ZF, SF, PF відповідно до результату:

```
AND призначення, джерело
AND EAX, EBX          ; EAX = EAX ∩ EBX
AND EAX, 0FFh         ; EAX = EAX ∩ 0xFF (залишаємо лише молодший байт)
AND EAX, 0FFFFFFF0h ; скидаємо молодші 4 біти в 0
```

Основні практичні застосування:

Маскування бітів – виділення потрібних розрядів через маску де 1 стоїть на позиціях які потрібно зберегти:

```
AND EAX, 00001111b  ; залишаємо лише молодші 4 біти
```

Перевірка окремого біта – якщо результат AND дорівнює нулю (ZF=1), біт не встановлений:

```
AND EAX, 00000001b  ; перевіряємо молодший біт: ZF=1 → біт 0, ZF=0 → біт 1
```

Скидання бітів – встановлення певних бітів у 0 через маску де 0 стоїть на потрібних позиціях:

```
AND EAX, 11111110b  ; скидаємо молодший біт в 0
```

OR – логічне АБО

Виконує побітову диз'юнкцію. Скидає CF і OF в 0, встановлює ZF, SF, PF:

```
OR призначення, джерело
```

```
OR EAX, EBX          ; EAX = EAX ∪ EBX
OR EAX, 00000001b  ; встановлюємо молодший біт в 1
```

Основне застосування – **встановлення бітів** через маску де 1 стоїть на позиціях, що потрібно встановити:

```
OR EAX, 10000000b  ; встановлюємо старший біт байта в 1
OR EAX, EBX          ; об'єднуємо набори прапорців двох регістрів
```

XOR – виключне АБО

Виконує побітове виключне або. Скидає CF і OF в 0, встановлює ZF, SF, PF:

```
XOR призначення, джерело
XOR EAX, EBX         ; EAX = EAX ⊕ EBX
XOR EAX, FFFFFFFFh ; інвертуємо всі біти EAX
```

Має кілька важливих практичних застосувань:

Інвертування бітів – XOR з маскою де 1 стоїть на потрібних позиціях:

```
XOR EAX, 00001111b   ; інвертуємо молодші 4 біти
```

Обнулення регістра – XOR регістра з самим собою завжди дає 0. Це класичний і ефективніший аналог MOV EAX, 0 оскільки займає менше байт у машинному коді:

```
XOR EAX, EAX      ; EAX = 0, і ZF = 1
```

Обмін значеннями без проміжного регістра – три послідовних XOR обмінюють значення двох регістрів:

```
XOR EAX, EBX      ; EAX = EAX ⊕ EBX
XOR EBX, EAX      ; EBX = EBX ⊕ (EAX ⊕ EBX) = початковий EAX
XOR EAX, EBX      ; EAX = (EAX ⊕ EBX) ⊕ початковий EAX = початковий
EBX
```

NOT – логічне заперечення

Виконує побітову інверсію операнда – замінює всі 0 на 1 і навпаки. Єдина логічна команда, що має лише один операнд і **не змінює жодного прапорця**:

```
NOT операнд
NOT EAX               ; інвертуємо всі біти EAX
NOT BYTE PTR [EBX] ; інвертуємо байт у пам'яті
```

Відмінність від NEG: NOT виконує побітову інверсію (обернений код), тоді як NEG виконує арифметичне заперечення (доповняльний код). Для числа 5:

NOT: 00000101 → 11111101**0** (побітова інверсія = -6 у доповняльному коді)

NEG: 00000101 → 11111101**1** (доповняльний код = -5)

TEST – перевірка бітів

Виконує побітову кон'юнкцію (AND) операндів, але **не зберігає результат** – лише встановлює прапорці ZF, SF, PF. Є логічним аналогом CMP:

```
TEST операнд1, операнд2
TEST EAX, EAX         ; перевіряємо чи EAX дорівнює нулю (ZF=1 якщо так)
TEST EAX, 00000001b ; перевіряємо молодший біт (ZF=1 якщо біт = 0)
TEST EAX, 10000000b ; перевіряємо старший біт байта
```

Відмінність від CMP: TEST використовується коли потрібно перевірити окремі біти, тоді як CMP використовується для порівняння числових значень. Класичні приклади застосувань:

```
TEST EAX, EAX     ; якщо ZF=1 → EAX = 0; якщо SF=1 → EAX від'ємне
TEST EAX, 1       ; якщо ZF=1 → EAX парне; якщо ZF=0 → EAX непарне
```

SHL / SAL – логічний і арифметичний зсув ліворуч

Зсувають біти операнда ліворуч на задану кількість позицій. Молодші розряди заповнюються нулями, старший біт, що виштовхується записується в CF. SHL і SAL є повністю ідентичними інструкціями – два різних імені для однієї машинної команди:

```
SHL операнд, кількість
SAL операнд, кількість
```

Кількість зсувів може бути константою або регістром CL:

```
SHL EAX, 1        ; EAX = EAX × 2
SHL EAX, 3        ; EAX = EAX × 8  (множення на 2³)
MOV CL, 4
SHL EAX, CL       ; EAX = EAX × 16 (кількість зсувів з регістра CL)
```

SHR – логічний зсув праворуч

Зсуває біти операнда праворуч. Старші розряди заповнюються нулями, молодший біт, що виштовхується записується в CF. Використовується для беззнакових чисел:

```
SHR операнд, кількість
```

```
SHR EAX, 1     ; EAX = EAX ÷ 2 (беззнакове)
SHR EAX, 2     ; EAX = EAX ÷ 4 (беззнакове)
```

SAR – арифметичний зсув праворуч

Зсуває біти операнда праворуч, але на відміну від SHR заповнює старші розряди **знаковим бітом,** а не нулями. Це зберігає знак числа і реалізує правильне ділення на степінь двійки для знакових чисел:

```
SAR операнд, кількість
MOV AX, -8 ; AX = -8 = 1111 1111 1111 1000
SAR AX, 1  ; AX = -4 = 1111 1111 1111 1100  (старший біт 1 зберігається)
SHR EAX, 1 ; AX = 32764 = 0111 1111 1111 1110
             (неправильно для знакового числа – старший біт замінено нулем)
```

Різниця між SHR і SAR принципова для від'ємних чисел – SHR дає неправильний результат при діленні знакового від'ємного числа.

ROL – циклічний зсув ліворуч

Зсуває біти ліворуч, але біт, що виходить з старшого розряду не губиться, а з'являється з молодшого розряду. Додатково старший біт копіюється в CF:

```
ROL операнд, кількість
MOV AX, -25001 ; AX = 1001 1110 0101 0111
ROL AX, 1      ; AX = 0011 1100 1010 1111  (старший біт перейшов у молодший)
```

ROR – циклічний зсув праворуч

Аналогічний ROL, але у зворотному напрямку – молодший біт переходить у старший розряд:

```
ROR операнд, кількість
MOV AX, -25001 ; AX = 1001 1110 0101 0111
ROR EAX, 1     ; AX = 1100 1111 0010 1011 (молодший біт перейшов у старший)
```

Циклічні зсуви ROL і ROR широко використовуються в криптографічних алгоритмах – зокрема SHA-256 і AES де ротація бітів є основною операцією перемішування даних. На відміну від логічних зсувів жоден біт не губиться, що є принциповою вимогою для оборотних криптографічних перетворень.

5.7. Організація переходів і циклів

Усі розглянуті раніше команди – арифметичні, логічні, пересилання даних – мають одну спільну рису: вони працюють з конкретними даними і виконуються суворо послідовно одна за одною. Процесор просто рухається по інструкціях зверху вниз, збільшуючи EIP після кожної з них.

Команди переходів і циклів принципово відрізняються – вони працюють не з даними, а з порядком виконання коду. Їх завдання змінити значення регістру EIP і таким чином передати керування в іншу частину програми. Саме ці команди перетворюють лінійну послідовність інструкцій на повноцінну програму з розгалуженнями і повтореннями – без них неможливо реалізувати жодну нетривіальну логіку.

З точки зору мов високого рівня команди переходів є апаратною основою на якій компілятор будує конструкції if, else, while, for і switch. Тобто будь-яка умовна конструкція або цикл у програмі на C в кінцевому підсумку перетворюється на одну або кілька команд переходу в машинному коді.

Безумовний перехід – JMP

JMP передає керування інструкції за вказаною міткою або адресою без жодних умов. Аналог оператора goto у мові C:

```
JMP мітка
    JMP skip          ; безумовно переходимо на мітку skip
    MOV EAX, 1        ; ця інструкціяніколи не виконається
skip:
    MOV EAX, 2        ; виконання продовжується тут
```

JMP може передавати керування як вперед так і назад по коду – саме перехід назад є основою організації циклів.

Умовні переходи

Умовні переходи перевіряють стан одного або кількох прапорців EFLAGS і передають керування на мітку лише якщо умова виконується. Якщо умова не виконується – виконання продовжується з наступної інструкції. Як правило умовний перехід використовується одразу після CMP або TEST.

При роботі з числами команди умовного переходу можна чітко поділяються на дві групи залежно від того чи є числа знаковими:

Для беззнакових чисел (перевіряють CF і ZF):

Команда	Синонім	Умова	Прапорці
JE	JZ	рівно (equal) / нуль	ZF=1
JNE	JNZ	не рівно / не нуль	ZF=0
JA	JNBE	більше (above)	CF=0, ZF=0
JAE	JNB	більше або рівно	CF=0
JB	JNAE	менше (below)	CF=1
JBE	JNA	менше або рівно	CF=1 або ZF=1

Для знакових чисел (перевіряють SF, OF, ZF):

Команда	Синонім	Умова	Прапорці
JE	JZ	рівно	ZF=1
JNE	JNZ	не рівно	ZF=0
JG	JNLE	більше (greater)	ZF=0, SF=OF
JGE	JNL	більше або рівно	SF=OF
JL	JNGE	менше (less)	SF≠OF
JLE	JNG	менше або рівно	ZF=1 або SF≠OF

Також є група команд умовних переходів за окремими прапорцями:

Команда	Умова
JC / JNC	CF=1 / CF=0
JO / JNO	OF=1 / OF=0
JS / JNS	SF=1 / SF=0
JP / JNP	PF=1 / PF=0

Реалізація конструкцій умовних переходів (код на мові С та аналог на мові Асемблера):

Конструкція if-else:

```
if (a > b)
    result = a;
else
    result = b;
```

```
__asm{
    MOV EAX, a
    MOV EBX, b
    CMP EAX, EBX     ; порівнюємо a і b
    JLE else_branch ; якщо a <= b ”перестрибуємо” if та переходимо до
else
    MOV result, EAX ; гілка if: result = a
    JMP end_if       ; ”перестрибуємо” гілку else
else_branch:
    MOV result, EBX ; гілка else: result = b
end_if:              ; мітка виходу з програми
}
```

Конструкція if без else:

```
if (a == 0)
    a = 1;
__asm{
    MOV EAX, a
    CMP EAX, 0
    JNE skip        ; якщо a ≠ 0 – пропускаємо тіло if
    MOV  EAX,  1        ;  записуємо 1 в змінну через регістр, оскільки
компілятор
    MOV a, EAX      ; не може визначити розмір операнда *
skip:               ;
}
```

* пересилання напряму можливо якщо вказати розмір константи, але можна явно вказати розмір і записати напряму:

```
MOV DWORD PTR a, 1 ;
```

Проте в реальних умовах складних програм, з метою максимізації швидкодії краще якомога довше тримати дані в регістрах і мінімізувати звертання до пам’яті.

Організація циклів

Розглянемо операції з циклами на прикладі простої програми: додавання чисел від 0 до 10. В асемблері замість складних конструкцій для організації циклів використовується звичайне порівняння лічильника та виконання умовного переходу на мітку.

Цикл з передумовою (while):

```
while (i < 10) {
    sum += i;
    i++;
}
__asm{
    MOV EAX, i
    MOV EBX, sum
while_start:
    CMP EAX, 10      ; перевіряємо умову на початку
    JGE while_end    ; якщо i >= 10 – виходимо
    ADD EBX, EAX     ; додаємо значення до результату  sum += i
    INC EAX          ; збільшуємо i:  i++
    JMP while_start  ; повертаємось на початок
while_end:
    MOV i, EAX
    MOV sum, EBX
}
```

Цикл з постумовою (do-while):

```
do {
    sum += i;
    i++;
} while (i < 10);
__asm{
    MOV EAX, i
    MOV EBX, sum
do_start:
    ADD EBX, EAX     ; sum += i
    INC EAX          ; i++
    CMP EAX, 10      ; перевіряємо умову в кінці
    JL do_start      ; якщо i < 10 – повторюємо
    MOV i, EAX
    MOV sum, EBX
}
```

Команда LOOP – організація циклу з лічильником

LOOP є спеціалізованою командою для організації циклів з лічильником. Автоматично зменшує ECX на 1 і передає керування на мітку якщо ECX ≠ 0. Якщо ECX = 0 – цикл завершується, а виконання програми продовжується з наступної по порядку інструкції:

```
мітка:
...
LOOP мітка
for (int i = 10; i > 0; i--)
    sum += i;
__asm{
    MOV ECX, 10      ; лічильник циклу
    MOV EAX, 0       ; sum = 0
loop_start:
    ADD EAX, ECX     ; sum += i
    LOOP loop_start  ; ECX--, якщо ECX ≠ 0 – повторити
    MOV sum, EAX
}
```

Варіанти команди LOOP:

Команда	Умова продовження
LOOP	ECX ≠ 0
LOOPE / LOOPZ	ECX ≠ 0 і ZF=1
LOOPNE / LOOPNZ	ECX ≠ 0 і ZF=0

LOOPE і LOOPNE дозволяють достроково завершити цикл якщо змінився певний прапорець – наприклад при пошуку елемента в масиві.

Реалізація циклу for

Цикл for можна реалізувати через збільшення регістра лічильника перевірку умови та умовний перехід:

```
for (int i = 0; i < 10; i++)
    sum += i;
__asm{
    MOV EAX, 0       ; sum = 0
    MOV ECX, 0       ; i = 0
```

```
for_start:
    CMP ECX, 10      ; i < 10?
    JGE for_end
    ADD EAX, ECX     ; sum += i
    INC ECX          ; i++
    JMP for_start
for_end:
    MOV sum, EAX
}
```

Або через команду LOOP, яка виконує всі 3 операції одночасно, але тоді лічильник іде у зворотному напрямку:

```
    MOV EAX, 0       ; sum = 0
    MOV ECX, 10      ; лічильник = 10
    MOV EBX, 0       ; i = 0
for_loop:
    ADD EAX, EBX     ; sum += i
    INC EBX          ; i++
    LOOP for_loop    ; ECX--, повторити якщо ECX ≠ 0
    MOV sum, EAX
```

Команди CALL і RET – виклик підпрограм

CALL передає керування підпрограмі зберігаючи на стеку адресу наступної інструкції – адресу повернення. RET знімає адресу повернення зі стека і передає туди керування:

```
    CALL my_func ; ESP -= 4, [ESP] = адреса повернення, перехід на my_func
next_row:        ; сюди повернеться керування після RET
    ...
my_func:
    ...          ; тіло підпрограми, довільний код
    MOV EAX, 42  ; результат виконання в функціях зазвичай записується в EAX
    RET          ; ESP += 4, перехід за адресою, що знята зі стека
```

У блоці __asm {} у Visual Studio CALL найчастіше використовується для виклику функцій C зі збереженням конвенції виклику. Але це вже тема окремого підрозділу про взаємодію асемблера з кодом C.

Переривання – INT

Команда INT генерує програмне переривання з заданим номером. У середовищі Windows більшість системних викликів виконуються не через INT безпосередньо, а через виклик функцій WinAPI, тому у блоці __asm {} INT використовується рідко. Проте варто знати, що саме через INT реалізована більшість системних функцій на рівні операційної системи:

```
INT 3     ; спеціальне переривання налагоджувача – точка зупинки
INT 21h   ; виклик функцій DOS (лише для 16-бітних програм)
```

Команда INT 3 є особливою – саме її використовує налагоджувач Visual Studio для встановлення програмних точок зупинки, замінюючи перший байт інструкції на код команди INT 3.

5.8. Співпроцесор x87 FPU

Співпроцесор x87 FPU (Floating Point Unit) – це окремий обчислювальний пристрій вбудований у процесор починаючи з Intel 486DX, що спеціалізується на виконанні арифметичних операцій з числами з плаваючою комою. До його появи операції з дійсними числами або емулювались програмно через цілочисельну арифметику, що було надзвичайно повільно, або виконувались окремим зовнішнім співпроцесором 8087, що підключався як окрема мікросхема. Назва x87 походить від позначень цих окремих співпроцесорів – 8087 (рис.5.6), 80287, 80387 –, що передували інтеграції в основний кристал.

Рисунок 5.6 – Співпроцесор Intel 8087-1 [21]

На відміну від регістрів загального призначення, що утворюють просто набір незалежних комірок, регістри x87 мають принципово іншу організацію (рис.5.7):

- вісім регістрів даних (R0 – R7), що організовані у вигляді кругового стека з восьми елементів ST(0)...ST(7), при чovy ST(0) – це завжди вершина стеку (TOP);

- п’ять службових регістрів: контролю, стану, тегів та два регістри вказівників.

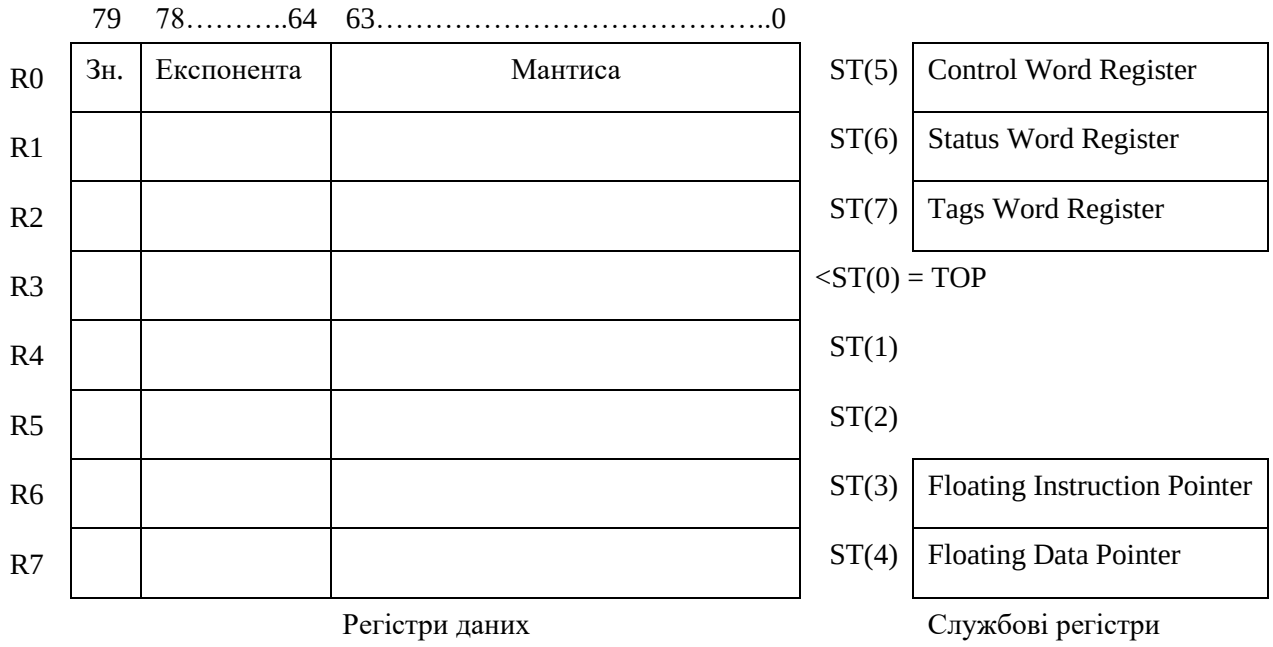

Рисунок 5.7 – Схема регістрів співпроцесора

Регістри даних

Принцип роботи з круговим стеком регістрів даних співпроцесора полягає в наступному:

Будь-яка команда завантаження даних співпроцесора автоматично переміщує вершину стеку співпроцесора: TOP = TOP + 1 (завантажене число стає в ST(0), а попереднє значення ST(0) зміщується в ST(1) і так далі). Наприклад, якщо в результаті виконання якоїсь команди вершиною стека став регістр R3 (рис.5.7), інші регістри розподіляються по колу: R4-ST(1), R5-ST(2), …, R2-ST(7). Це її є їх поточні імена ST(i), i=1,…,7 на момент виконання даної команди співпроцесора. Звертатися ж напряму до регістрів R0-R7 не можна. Якщо в цих регістрах є дані, то вони можуть слугувати операндами в командах співпроцесора.

Окрім восьми регістрів даних x87 розглянемо службові регістри:

Слово стану (Status Word) – 16-бітний регістр (рис. 5.8), що відображає поточний стан співпроцесора (тобто є аналогом регістру FLAGS):

Біт	Назва	Призначення
0	IE	помилка недійсної операції
2	ZE	ділення на нуль
3	OE	переповнення
4	UE	підтікання
8–10	TOP	номер регістра, що є поточною вершиною стека
14	B	співпроцесор зайнятий

Біти C0, C1, C2, C3 (біти 8, 9, 10, 14) – прапорці умов, що встановлюються командою порівняння FCOM і використовуються для умовних переходів.

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

Рисунок 5.8 – Регістр стану співпроцесора

Розглянемо наявні прапорці:

Прапорці умов (станів). Conditions

C0 – Аналог прапорця переносу CF процесора. Містить ознаки переносу інформації зі старшого біта.

C1 – Прапорець умови співпроцесора. Не є аналогічним прапорцю умови процесора, а пов’язаний з прапорцем SE.

C2 – Аналог прапорці парності PF процесора.

C3 – Аналог прапорця нуля ZF процесора. Встановлюється в 1, якщо результат виконання команди дорівнює 0.

Прапорці виключень. Exceptions

ES – Загальний прапорець помилки. Встановлюється в 1, якщо виникла хоча б одна немаскована помилка.

SE – Помилка стеку. Якщо C1=1, то виникло переповнення стеку (команда намагалась записати інформацію в не порожню комірку стеку). Якщо C1=0, то виникло антипереповнення стеку (команда намагались зчитати інформацію з пустої комірки стеку).

PE (precision) – Прапорець втрати точності. Встановлюється коли результат не може бути представлений точно, наприклад 1/3.

UE (Underflow) – Прапорець антипереповнення – результат занадто малий.

OE (Overflow) – Прапорець переповнення – результат завеликий.

ZE (Zero divide) – Прапорець ділення на нуль.

DE (Denormal) – Прапорець денормалізованого операнда – виконано операцію над денормалізованим числом.

IE (Invalid operation) – Прапорець недопустимої операції.

Інші прапорці станів

B (Busy) – Прапорець занятості FPU. Використовується для забезпечення сумісності з i8087 та i80287. В сучасних співпроцесорах його значення співпадає із значенням прапорця ES.

TOP – Двійкове число від 000 до 111, котре вказує, який з регістрів R0-R7 в даний момент є вершиною стеку ST.

Варто звернути увагу, що у співпроцесора немає аналога прапорця SF (sign flag).

Слово керування (Control Word) – 16-бітний регістр, що керує режимами роботи співпроцесора (рис. 5.9). Молодші біти слова управління визначають маскування (заборону) обчислювальних виключень. Біти 0-5 містять індивідуальні маски для кожного із шести виключень, які визначаються співпроцесором при виконанні операцій з плаваючою комою. Старші біти (8-12) управляючого слова визначають параметри роботи співпроцесора.

Біти	Призначення
0–5	маски виключень (дозволяють або забороняють генерацію переривань при помилках)
8–9	точність обчислень (24, 53 або 64 біти мантиси)
10–11	режим округлення (до найближчого, вниз, вгору, до нуля)

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

Рисунок 5.9 – Регістр керування процесора

Слово тегів (Tag Word) – 16-бітний регістр, що описує стан кожного з восьми регістрів стека (рис.5.10): чи містить він дійсне число, нуль, спеціальне значення або є порожнім:

Значення тегу	Значення
00	Валідне (нормальне) число
01	Нуль
10	Спеціальне значення (NaN, нескінченність)
11	Регістр порожній (Empty)

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

Рисунок 5.10 – Регістр тегів

Інформація регістра тегів може використовуватись програмами, які оброблюють виключення, що виникають при обчисленнях у співпроцесорі

Регістри вказівників (IP та DP).

Це службові регістри (Instruction Pointer та Data Pointer), що зберігають адресу останньої виконаної команди x87 та адресу даних у пам'яті, з якими вона працювала та використовуються переважно операційною системою для обробки помилок (exception handling), щоб знати, де саме в коді стався збій.

Співпроцесор може виконувати операції з сімома звичайними типами даних, а також із спеціальними числами. Звичайні дані можуть бути дійсними або цілими числами із знаком. Наявність цілочисельних даних дозволяє виконувати так звані змішані обрахування, коли в якості операндів

використовуються і цілі, і дійсні числа. Співпроцесор виконує всі обрахування в 80-бітному розширеному дійсному форматі, а 16-, 32- і 64-бітні дані використовуються для обміну даними і можуть застосовуватись в командах співпроцесора як операнди

Типи звичайних даних співпроцесора:

Тип даних	**Довжина (біт)**	**Кількість значимих цифр**	**Діапазон представлення**
Ціле слово	16	4-5	-32768…32767
Коротке ціле	32	9-10	-2147483648 … - 2147483647
Довге ціле	64	18-19	9223372036854775808 … 9223372036854775807
Упаковане двійководесяткове	80	18	-999999999999999999 … 999999999999999999
Коротке дійсне	32	7-8	1.175494351e-38 … 3.402823466e+38
Довге дійсне	64	15-16	2.2250738585072014e-308... 1.7976931348623158e+308
Розширене дійсне	80	19-20	3.3621031431120935063e4932… 1.18973149535723176 5e+4932

Крім звичайних чисел, специфікація стандарту IEEE 754 передбачає декілька спеціальних форматів, які можуть виникнути в результаті виконання математичних операцій співпроцесора і над якими можна виконувати окремі операції.

• Додатний нуль – всі біти числа скинуті до нуля.

• Від’ємний нуль – знаковий біт дорівнює 1, всі інші біти числа скинуті до нуля.

• Додатна нескінченність – знаковий біт дорівнює 0, всі біти експоненти установлені в 1, а біти мантиси скинуті до нуля.

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

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

• Невизначеність – знаковий біт дорівнює1, перший біт мантиси дорівнює 1 (для 80- розрядних чисел перші два біта дорівнюють 11), інші біти мантиси скинуті до нуля, всі біти експоненти встановленні в 1.

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

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

• Непідтримуване число – всі інші ситуації.

Основні команди співпроцесора x87.

Команди x87 починаються з літери «**F**» , що відрізняє їх від команд основного процесора.

Завантаження даних у стек:

Команда	**Дія**
FLD src	завантажити дійсне число з пам'яті на вершину стека ST(0)
FILD src	завантажити ціле число з пам'яті на ST(0), автоматично перетворити на дійсне
FLDZ	завантажити константу 0.0
FLD1	завантажити константу 1.0
FLDPI	завантажити константу π
FLDL2E	завантажити $\log_2(e)$
FLDL2T	завантажити $\log_2(10)$

Збереження даних зі стека:

Команда	**Дія**
FST dst	зберегти ST(0) в пам'ять без виштовхування зі стека
FSTP dst	зберегти ST(0) в пам'ять і виштовхнути зі стека (pop)
FIST dst	зберегти ST(0) як ціле число в пам'ять
FISTP dst	зберегти ST(0) як ціле число і виштовхнути зі стека

Арифметичні операції

Усі арифметичні команди x87 працюють з вершиною стека ST(0) як одним з операндів:

Команда	Дія
FADD src	ST(0) = ST(0) + src
FSUB src	ST(0) = ST(0) − src
FMUL src	ST(0) = ST(0) × src
FDIV src	ST(0) = ST(0) ÷ src
FADDP	ST(1) = ST(1) + ST(0), виштовхнути ST(0)
FSUBP	ST(1) = ST(1) − ST(0), виштовхнути ST(0)
FMULP	ST(1) = ST(1) × ST(0), виштовхнути ST(0)
FDIVP	ST(1) = ST(1) ÷ ST(0), виштовхнути ST(0)
FIADD src	ST(0) = ST(0) + ціле число src
FISUB src	ST(0) = ST(0) − ціле число src
FIMUL src	ST(0) = ST(0) × ціле число src
FIDIV src	ST(0) = ST(0) ÷ ціле число src

Якщо після команди не вказується операнд-джерело “src” – операція виконується між ST(0) та ST(1). Суфікс **«P»** означає, що після операції значення регістру ST(0) виштовхується зі стека. Це дозволяє виконати операцію і одразу звільнити регістр.

Математичні функції:

Команда	Дія
FSQRT	ST(0) = √ST(0)
FABS	ST(0) = |ST(0)| (абсолютне значення)
FCHS	ST(0) = −ST(0) (зміна знаку)
FSIN	ST(0) = sin(ST(0))
FCOS	ST(0) = cos(ST(0))
FPTAN	обчислює tan(ST(0))
FPATAN	обчислює arctg(ST(1)/ST(0))
F2XM1	ST(0) = 2^ST(0) − 1
FYL2X	ST(1) = ST(1) × $\log_2$(ST(0))

Команди порівняння:

Команда	Дія
FCOM src	порівняти ST(0) з src, встановити біти C0-C3
FCOMP src	порівняти і виштовхнути ST(0)
FCOMPP	порівняти ST(0) і ST(1), виштовхнути обидва
FTST	порівняти ST(0) з нулем
FUCOM	порівняння з обробкою NaN

Після FCOM результат порівняння знаходиться в бітах C0 і C3 слова стану. Для використання з командами умовного переходу необхідно перенести слово стану в регістр AX через FNSTSW і перевірити відповідні біти:

```
FCOM          ; порівнюємо ST(0) і ST(1)
FNSTSW AX     ; переносимо слово стану в AX
SAHF          ; переносимо AH в EFLAGS
JA  label     ; тепер можна використовувати звичайні умовні переходи
```

Службові команди:

Команда	Дія
FINIT	ініціалізація співпроцесора
FXCH ST(n)	обмін ST(0) і ST(n) місцями
FFREE ST(n)	позначити регістр як порожній
FNSTSW dst	зберегти слово стану в пам'ять або AX
FLDCW src	завантажити слово керування
FNSTCW dst	зберегти слово керування

Перед використанням співпроцесора обов’язково потрібно його ініціалізувати командою **FINIT**.

Розглянемо приклад – обчислення виразу (a + b) × c

```
float a = 2.5f, b = 1.5f, c = 4.0f, result;
```

```
__asm {
    FINIT           ; ініціалізуємо співпроцесор
    FLD a           ; ST(0) = a = 2.5
    FLD b           ; ST(0) = b = 1.5, ST(1) = a = 2.5
    FADDP           ; ST(0) = ST(1) + ST(0) = 4.0, стек: ST(0) = 4.0
    FLD c           ; ST(0) = c = 4.0, ST(1) = 4.0
    FMULP           ; ST(0) = ST(1) × ST(0) = 16.0
    FSTP result     ; result = 16.0, стек порожній
}
```

Покроковий стан стека:

1. після FINIT: стек порожній
2. після FLD a: ST(0)=2.5
3. після FLD b: ST(0)=1.5 ST(1)=2.5
4. після FADDP: ST(0)=4.0
5. після FLD c: ST(0)=4.0 ST(1)=4.0
6. після FMULP: ST(0)=16.0
7. після FSTP: стек порожній, result=16.0

Формати даних у пам'яті

Команди співпроцесора x87 підтримують кілька форматів операндів у пам'яті:

Тип	**Розмір**	**Команди**
short int	16 біт	FILD, FIST, FISTP
int	32 біти	FILD, FIST, FISTP
long int	64 біти	FILD, FIST, FISTP
float	32 біти	FLD, FST, FSTP
double	64 біти	FLD, FST, FSTP
long double	80 біт	FLD, FST, FSTP

Тип операнда у пам'яті визначається типом змінної в мові C, що використовується як операнд: асемблер автоматично генерує правильну команду:

```
float  f = 1.5f;     // 32-бітний float
double d = 1.5;      // 64-бітний double
int    i = 5;        // 32-бітне ціле
```

```
__asm {
    FLD  f      ; завантажує 32-бітний float
    FLD  d      ; завантажує 64-бітний double
    FILD i      ; завантажує 32-бітне ціле і перетворює на дійсне
}
```

5.9. Інструменти та практичне застосування Асемблеру

Незважаючи на те, що сучасні компілятори генерують надзвичайно ефективний машинний код, асемблер залишається незамінним у кількох областях:

Системне програмування – ядра операційних систем містять фрагменти на асемблері для операцій, що принципово неможливо виразити мовою C: перемикання між процесами, обробка переривань, ініціалізація процесора при завантаженні, керування привілейованими регістрами. Наприклад ядро Linux містить тисячі рядків асемблерного коду саме для цих цілей.

Вбудовані системи – мікроконтролери з обмеженими ресурсами, в процесі роботи яких важливим є кожен байт пам'яті та кожен такт процесора. Програмування мікроконтролерів на асемблері дозволяє точно контролювати часові характеристики і розмір коду.

Криптографія – реалізації алгоритмів шифрування (AES, SHA) часто містять асемблерні оптимізації з використанням спеціалізованих наборів інструкцій SSE/AVX, що дозволяють обробляти кілька блоків даних одночасно.

Драйвери пристроїв – взаємодія з апаратним забезпеченням на низькому рівні, пряме керування портами введення-виведення і регістрами пристроїв.

Антивірусне програмне забезпечення і кібербезпека – аналіз шкідливого програмного забезпечення, пошук вразливостей, дослідження закритого програмного забезпечення – все це вимагає вміння читати і розуміти асемблерний код.

Реверс-інжиніринг – процес аналізу готового програмного або апаратного продукту з метою виявлення його архітектури, логіки функціонування, структури та принципів роботи без доступу до вихідних матеріалів (вихідного коду чи схем). У програмному контексті реверс-інжиніринг зазвичай стосується аналізу виконуваних файлів (EXE, DLL, BIN), а також прошивок і драйверів. Основним інструментом цього аналізу є дизасемблювання або декомпіляція.

Мікрооптимізація коду – цілеспрямована оптимізація окремих критичних ділянок коду на рівні асемблерних інструкцій. Важливо розуміти, що мікрооптимізація виправдана лише після профілювання програми і виявлення вузьких місць, оскільки оптимізація коду, що і так добре працює, рідко має сенс.

Розглянемо основні приклади оптимізації коду:

- Заміна множення зсувом.

Множення на степінь двійки замінюється логічним зсувом, що виконується за один такт замість кількох:

```
IMUL EAX, 8; Повільніше
SHL EAX, 3 ; Швидше, проте EAX × 2³ = EAX × 8
```

Сучасні компілятори виконують цю оптимізацію автоматично, але для складніших випадків, наприклад множення на 10 можна використати LEA:

```
LEA EAX, [EAX + EAX*4] ; EAX = EAX + EAX×4 = EAX × 5
SHL EAX, 1             ; EAX × 5 × 2 = EAX × 10
```

Команди LEA та SHL виконуються процесором надзвичайно швидко (зазвичай за 1 такт). А більш складна команда множення MUL або, ще гірше, IMUL, може займати значно більше процесорного часу.

- Обнулення регістра

```
MOV EAX, 0    ; Займає більше байт у машинному коді
XOR EAX, EAX  ; Компактніше і швидше
```

- Перевірка на нуль

```
CMP EAX, 0    ; Два операнди – порівняння з константою
JE  label
TEST EAX, EAX ; Один операнд – те саме, але компактніше
JZ  label
```

- Перевірка парності

Наприклад перевірка чи число парне через AND:

```
TEST EAX, 1 ; перевіряємо молодший біт, якщо 0 отримаємо прапорець нуля ZF=1,
JZ even     ; відповідно число парне
```

- Інкремент і декремент замість ADD/SUB

Зазначені в прикладі команди еквівалентні, але інкремент та декремент компактніші при зберіганні в пам’яті, оскільки мають всього 1 операнд.

```
ADD EAX, 1  =  INC EAX
SUB EAX, 1  =  DEC EAX
```

- Уникнення залежностей між інструкціями

Сучасні процесори виконують інструкції паралельно, якщо вони не залежать одна від одної. Код, що враховує це виконується швидше:

```
ADD EAX, EBX ; такт 1
ADD EAX, ECX ; такт 2
ADD EAX, EDX ; такт 3
```

Код займе 3 такти, оскільки кожна інструкція залежить від попередньої. Замінивши порядок додавання на різні регістри ми отримаємо дві незалежні операції, що можуть виконуватись паралельно, і код займе 2 такти:

```
ADD EAX, EBX ; такт 1, поток 1
ADD ECX, EDX ; такт 1, поток 2
ADD EAX, ECX ; такт 2
```

- Мінімізація звернень до пам'яті

Звернення до пам'яті навіть через кеш першого рівня займає кілька тактів, тоді як операції з регістрами – один такт. Тому варто максимально довго тримати дані в регістрах:

Поганий приклад – багато звернень до пам'яті

```
MOV EAX, counter
INC EAX
MOV counter, EAX
MOV EAX, counter
ADD EAX, value
MOV counter, EAX
```

Оптимізований аналог – одне читання і один запис:

```
MOV EAX, counter
INC EAX
ADD EAX, value
MOV counter, EAX
```

Перегляд асемблерного коду у Visual Studio

Visual Studio дозволяє переглядати асемблерний код, що генерується компілятором для програми написаної на С – це надзвичайно корисний інструмент для розуміння як компілятор оптимізує код і для пошуку вузьких місць.

Для перегляду асемблерного коду під час налагодження:

1. Поставити точку зупинки;
2. Запустити програму в режимі налагодження (F5);
3. Коли виконання зупиниться – відкрити Debug → Windows → Disassembly (Alt+8);

У вікні дизасемблера відображається асемблерний код з адресами інструкцій, машинними кодами і відповідними рядками коду С:

Наприклад команда int result = a + b;

```
00401234  mov         eax, dword ptr [a]
00401237  add         eax, dword ptr [b]
0040123A  mov         dword ptr [result], eax
```

Це дозволяє:

- побачити, що саме генерує компілятор для конкретного коду С
- порівняти ефективність різних способів запису одного і того ж виразу
- перевірити чи компілятор застосував очікувані оптимізації
- зрозуміти чому певна ділянка коду працює повільно

Дизасемблювання

Дизасемблювання – це процес зворотній до асемблювання, а саме – перетворення машинного коду (бінарного виконуваного файлу) в «людиночитабельне» текстове представлення мовою асемблера. Цей інструмент має велике значення для розуміння, як працює програма «всередині», а також для відлагодження, оптимізації, навчання та реверс-інжинірингу і в першу чергу використовується для аналізу програм, вихідний код яких недоступний. Яким може бути практичне застосування дизасемблювання?

- **Аналіз шкідливого програмного забезпечення** – дизасемблювання вірусів і троянів дозволяє зрозуміти їх поведінку, знайти індикатори компрометації і розробити засоби захисту.

- **Пошук вразливостей** – аналіз бінарних файлів на наявність помилок безпеки: переповнення буфера, неправильна обробка вхідних даних, некоректна робота з пам'яттю.
- **Зворотна розробка протоколів** – аналіз закритих мережевих протоколів для забезпечення сумісності або дослідження безпеки.
- **Перевірка роботи компілятора** – як вже зазначалось, перегляд асемблерного коду дозволяє перевірити чи компілятор правильно оптимізує код і виявити неочікувану поведінку.

Розглянемо декілька основних застосунків, що довзоляють проводити дизасемблювання:

Вбудований дизасемблер Visual Studio, що розглядався вище є найпростішим інструментом і доступний безпосередньо під час налагодження.

IDA Pro – найпотужніший професійний дизасемблер, що використовується в індустрії кібербезпеки. Автоматично визначає функції, відновлює типи даних, будує граф потоку керування і підтримує сотні процесорних архітектур. Має безкоштовну версію IDA Free з обмеженими можливостями.

Ghidra – безкоштовний дизасемблер і декомпілятор розроблений АНБ США і відкритий у 2019 році. Окрім дизасемблювання містить декомпілятор, що намагається відновити псевдокод мовою C з машинного коду – це суттєво спрощує аналіз складних програм.

x64dbg – безкоштовний налагоджувач з вбудованим дизасемблером для динамічного аналізу – тобто аналізу програми під час її виконання. Дозволяє встановлювати точки зупинки, змінювати значення регістрів і пам'яті у реальному часі.

Важливо звертати увагу на правові та етичні аспекти реверс-інжинірингу та дизасемблювання. Хоча дизасемблювання програмного забезпечення є технічно можливим, але не завжди правомірним. Більшість ліцензійних угод забороняють зворотну розробку комерційного програмного забезпечення (реверс-інжиніринг). Легальне використання дизасемблювання включає аналіз власного коду, дослідження безпеки у межах програм bug bounty, аналіз шкідливого програмного забезпечення і академічні дослідження. Перед дизасемблюванням будь-якого програмного забезпечення необхідно переконатись у правомірності таких дій.

Висновки до розділу 5

Асемблер – це програма, що перетворює текстовий файл написаний мовою асемблера на машинний код, причому, кожен рядок тексту відповідає рівно одній машинній інструкції.

Мова асемблера – це низькорівнева мова програмування, яка, на відміну від мов високого рівня, зокрема мови C, дозволяє безпосередньо працювати з регістрами, пам’яттю та апаратними ресурсами комп’ютера, що гарантує програмісту повний контроль над апаратними ресурсами. Так використання

асемблерних вставок дозволяє оптимізувати виконання певних обчислень, отримуючи максимальну продуктивність.

Архітектура IA-32 базується на використанні регістрів, які є надшвидкими комірками пам'яті всередині процесора. Ключову роль відіграють регістри загального призначення (EAX, EBX, ECX, EDX та ін.), лічильник команд (EIP), що визначає наступну інструкцію, та регістр прапорців (EFLAGS), який відображає стан обчислень і керує режимами роботи.

Стек є критично важливою структурою даних, що працює за принципом LIFO («останній прийшов — перший вийшов») і росте в бік менших адрес. Він використовується для тимчасового збереження значень регістрів, передачі аргументів функціям та зберігання адрес повернення.

Система команд асемблера охоплює широкий спектр операцій, включаючи пересилання даних (MOV, XCHG), арифметичні обчислення (ADD, SUB, MUL, DIV), логічні операції (AND, OR, XOR, NOT) та зсуви бітів. Особливе значення мають команди порівняння (CMP, TEST) та умовних переходів, які дозволяють реалізувати складну логіку розгалужень та циклів (if-else, while, for).

Для роботи з числами з плаваючою комою використовується співпроцесор x87 FPU, який має власну стекову організацію з восьми 80-бітних регістрів (ST(0)–ST(7)). Всі обчислення в ньому виконуються з підвищеною точністю, незалежно від формату вхідних даних.

Сучасне застосування асемблера зосереджене у специфічних нішах, де критично важливі продуктивність та прямий доступ до «заліза». До них належать системне програмування (ядра ОС, драйвери), розробка вбудованих систем, криптографія, реверс-інжиніринг та мікрооптимізація критичних ділянок коду.

Питання до самоконтролю

1. У чому полягає принципова відмінність між машинним кодом та мовою асемблера?
2. Дайте визначення поняттям «мнемоніка» та «асемблер».
3. Чому мова асемблера вважається непереносимою мовою програмування?
4. Які основні області застосування асемблера в сучасному програмуванні?
5. Поясніть роль регістра EIP (лічильника команд) у роботі процесора. Чи може програміст змінювати його безпосередньо?
6. Назвіть регістри загального призначення та опишіть специфічне функціональне призначення кожного з них.
7. До яких частин регістрів EAX, EBX, ECX та EDX можна звертатись відокремлено?
8. Які сегментні регістри існують в архітектурі IA-32, і за що відповідає кожен із них?
9. Що таке «плоска модель пам'яті» і як вона реалізована в сучасних ОС?
10. Для чого використовується регістр прапорців (EFLAGS)?

11. Опишіть призначення основних прапорців стану: ZF, CF, SF та OF. У яких випадках вони встановлюються?
12. Яка різниця між прапорцями CF (Carry Flag) та OF (Overflow Flag)?
13. Як працює прапорець напрямку DF і на які команди він впливає?
14. Поясніть роль прапорця пастки (TF) у процесі налагодження програм.
15. Принципи функціонування стеку.
16. Поясніть принцип LIFO та особливість росту стека в архітектурі IA-32.
17. Які дії виконують команди PUSH та POP з регістром ESP?
18. Чому при збереженні кількох регістрів у стек їх відновлення має відбуватися у зворотному порядку?
19. Яка різниця в порядку операндів між синтаксисом Intel та AT&T?
20. Опишіть основні режими адресації операндів (регістрова, безпосередня, пряма, непряма).
21. Для чого використовуються команди MOVSX та MOVZX?
22. Поясніть поняття порядку байтів Little-endian та роль команди BSWAP.
23. Які регістри є прихованими операндами для команд множення (MUL) та ділення (DIV)?
24. Поясніть різницю між командами CMP та SUB?
25. Як працює команда LOOP і який регістр вона використовує як лічильник?
26. Яка різниця між логічним (SHR) та арифметичним (SAR) зсувом праворуч?
27. Для чого використовується співпроцесор та оптимізація
28. Як організовані регістри даних у співпроцесорі x87 FPU? Що таке вершина стека ST(0)?
29. Які основні методи мікрооптимізації коду можна реалізувати за допомогою асемблера (наприклад, обнулення регістрів чи заміна множення)?
30. Що таке дизасемблювання та які інструменти для цього існують?

СПИСОК ВИКОРИСТАНИХ ТА РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. von Neumann J. First draft of a report on the EDVAC. Moore School of Electrical Engineering University of Pennsylvania. 1945. pp.110. – URL: https://archive.org/details/20200901-vN_First_Draft_Report_EDVAC_Moore_Sch_1945/page/n1/mode/2up
2. ISO/IEC 2382:2015 – Information technology. Vocabulary – URL: https://www.iso.org/standard/63598.html
3. ISO/IEC/IEEE 24765 – Systems and Software Engineering. Vocabulary. – URL: https://www.iso.org/standard/71952.html
4. Advanced Configuration and Power Interface (ACPI) Specification. Release 6.6. – URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf
5. PCI-SIG, спільнота, відповідальна за розробку та підтримку стандартизованого підходу до передачі даних вводу/виводу периферійних компонентів. – URL: https://pcisig.com/
6. USB Implementers Forum, Inc. – URL: https://www.usb.org/
7. IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2019 (Revision of IEEE 754-2008)* , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229
8. ISO 6093:1985 Information processing – Representation of numerical values in character strings for information interchange. – URL: https://www.iso.org/standard/12285.html
9. ISO/IEC 10646:2020 – Information technology – Universal coded character set (UCS). – URL: https://www.iso.org/standard/76835.html
10. H. Burderer. Technical Marvels. Communications of the ACM. 2024. URL: https://www.researchgate.net/publication/379958472_Technical_Marvels_1
11. Babbage's Analytical Engine, 1834-1871. (Trial model). – Science Museum Group Collection. – URL: https://collection.sciencemuseumgroup.org.uk/objects/co62245/babbages-analytical-engine-1834 1871-trial-model-analytical-engines
12. ENIAC (Electronic Numerical Integrator and Computer), c. 1946. – Encyclopædia Britannica. – URL: https://www.britannica.com/technology/ENIAC#/media/1/183842/203028
13. The IBM 1401. – IBM history. – URL: https://www.ibm.com/history/1401
14. A look at IBM S/360 core memory: In the 1960s, 128 kilobytes weighed 610 pounds. – Ken Shirriff's blog. – URL: https://www.righto.com/2019/04/a-look-at-ibm-s360-core-memory-in-1960s.html
15. The Intel ® 8086 and the IBM PC. – URL: https://www.intel.com/content/www/us/en/history/virtual-vault/articles/the-8086-and-the-ibm-pc.html
16. Quantum computer. Google Quantum AI – URL: https://quantumai.google/quantumcomputer

17. The TOP500 list project. – URL: https://top500.org/
18. ISO/IEC 60559:2020 Information technology – Microprocessor Systems – Floating-Point arithmetic. https://www.iso.org/standard/80985.html
19. Синтаксис AT&T. – URL: https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_AT%26T
20. The curse of AT&T and Intel assembly syntax for x86-64 programmers – URL: https://www.reddit.com/r/asm/comments/13y872l/the_curse_of_att_and_intel_assembly_syntax_for
21. Intel 8087-1 CDIP. – https://www.hardware-collection.net/en/fpu/Intel-8087-1-CDIP-cpu-no3658.html
22. x86 Assembly Guide. University of Virginia Computer Science CS216: Program and Data Representation, Spring 2006. – URL: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
23. Samuel Yang Basic Execution Environment of Intel Processor 32-bit Architecture. July 30, 2018. URL: https://grandidierite.github.io/basic-execution-environment-of-intel-processor-32-bit-architecture/
24. Vector Processor. URL: https://electronicsdesk.com/vector-processor.html
25. An Overview of Cluster Computing. URL: https://www.geeksforgeeks.org/computer-networks/an-overview-of-cluster-computing/
26. High Bandwidth Memory. – URL: https://en.wikipedia.org/wiki/High_Bandwidth_Memory
27. Computer architecture and security: fundamentals of designing secure computer systems / Shuangbao (Paul) Wang, Robert S. Ledley. 2013
28. Sarah Harris, David Harris. Digital Design and Computer Architecture. ARM Edition. Morgan Kaufmann. 2016 – 584p.
29. Мельник А. О. Архітектура комп'ютера: підручник. – Луцьк : Волинська обласна друкарня, 2008. – 470 с.
30. Архітектура комп’ютерів. Особливості використання комп’ютерів в ІС : навчальний посібник / С. В. Кавун, І. В. Сорбат. – Харків : Вид. ХНЕУ, 2010. – 256 с.
31. John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach. 6th Edition. Morgan Kaufmann. 2017 – 936 p.
32. ДСТУ ISO/IEC 2382:2017 Інформаційні технології. Словник термінів (ISO/IEC 2382:2015, IDT).
33. David Harris, Sarah Harris. Digital Design and Computer Architecture. 2nd Edition. Morgan Kaufmann. 2012 – 720p.
34. Тарарака В.Д. Архітектура комп'ютерних систем: навчальний посібник. – Житомир : ЖДТУ, 2018. – 383 с.
35. Jim Ledin, Dave Farley. Modern Computer Architecture and Organization. Packt Publishing Ltd, 2022 - Computers – 560 p. – URL: https://viterbi-

web.usc.edu/~yudewei/main/sources/books/Modern%20Computer%20Architecture%20and%20Organization%20Learn%20processor%20architecture%20including%20RISC-V,%20and%20design%20of%20PCs,%20cloud%20servers,...%20(Jim%20Ledin)%20(z-lib.org).pdf

36. Patricio Bulić. Understanding Computer Organization: A Guide to Principles Across RISC-V, ARM Cortex, and Intel Architectures. Undergraduate Topics in Computer Science. - Springer Cham. - 2024. - 297 p. URL: https://doi.org/10.1007/978-3-031-58075-8
37. Shuangbao Paul Wang. Computer Architecture and Organization: Fundamentals and Architecture Security. Fundamentals and Architecture Security. - Springer Singapore. – 2021. – 337 p. URL: https://doi.org/10.1007/978-981-16-5662-0
38. Bernard Goossens. Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis. Undergraduate Topics in Computer Science. – 2023. – 439 p. https://doi.org/10.1007/978-3-031-18023-1
39. Матвієнко, М.П. Архітектура комп’ютерів: навч. посібник / М.П, Матвієнко, В.П, Розен, О.М. Закладний. – К.: Видавництво Ліра-К, 2020. – 264 с.
40. Кравченко Ю.В., Лещенко О.О., Герасименко О.Ю., Труш О.В., Дахно Н.Б. Архітектура комп’ютера. Частина 1: навчальний посібник. – Київ: КНУ імені Тараса Шевченка, 2022. – 216 с.

ДЛЯ НОТАТОК

Навчальне видання

ШЕЛУХА Олексій Олегович

ОСНОВИ АРХІТЕКТУРИ КОМП’ЮТЕРА

Навчальний посібник

Електронне видання

Редактори
Комп’ютерна верстка Шелуха О. О.

Гарнітура Times New Roman, Courier New, Cambria Math
Ум. друк. арк. 24,18

Державний університет «Житомирська політехніка»
м. Житомир, вул. Чуднівська, 103