Джим Ледін

Сучасна архітектура та організація комп'ютерів

2-ге видання

2026

Зміст

Передмова **8**

Укладачі **12**

Про автора 12
Про рецензентів 12

Вступ **14**

Для кого ця книга 15
Як організована ця книга 15
Як отримати максимальну віддачу від цієї книги 19
Завантажте файли з прикладами коду 19
Завантажте кольорові зображення 19
Умовні позначення та угоди 19
Зв'яжіться з нами 20
Поділіться своїми думками 21

Розділ 1.Введення в архітектуру комп'ютерів **22**

Технічні вимоги 23
Еволюція автоматичних обчислювальних пристроїв 23
Аналітична машина Чарльза Беббіджа 23
ENIAC 26
IBM PC 27
iPhone 31
Закон Мура 32
Архітектура комп'ютерів 35
Представлення чисел рівнями напруги 36
Двійкові та шістнадцяткові числа 36
Микропроцессор 6502 40
Набір інструкцій мікропроцесора 6502 43
Резюме 46
Вправи 47

Розділ 2.Цифрова логіка **49**

Технічні вимоги 50

Електричні схеми ... 50
Транзистор ... 51
Логічні вентилі ... 52
Засувки ... 57
Тригери ... 60
Регістри ... 62
Суматори ... 64
Затримка поширення ... 65
Синхронізація ... 67
Послідовна логіка ... 68
Мови опису апаратних засобів ... 69
VHDL ... 69
Резюме ... 75
Вправи ... 75

Розділ 3.Елементи процесора ... 76

Технічні вимоги ... 76
Простий процесор ... 77
Пристрій керування ... 78
Арифметико-логічний пристрій ... 81
Регістри ... 86
Набір інструкцій процесора ... 88
Режими адресації ... 89
Режим безпосередньої адресації ... 89
Режим абсолютної адресації ... 90
Режим непрямої індексної адресації ... 91
Режими непрямої індексної адреси ... 93
Категорії інструкцій ... 95
Інструкції завантаження та збереження ... 95
Інструкції передавання даних із регістра в регістр ... 96
Інструкції стека ... 96
Арифметичні інструкції ... 96
Логічні інструкції ... 97
Інструкції розгалуження ... 98
Інструкції виклику підпрограми та повернення з підпрограми ... 98
Інструкції для роботи з прапорами процесора ... 99
Інструкції для роботи з перериваннями ... 99
Інструкція відсутності операцій ... 100
Обробка переривань ... 100
Обробка ***IRQ*** ... 100
Обробка ***NMI*** ... 102

Обробка інструкції BRK ... 102
Операції введення-виведення ... 104
Програмоване введення-виведення ... 106
Введення-виведення з керуванням за перериваннями ... 106
Прямий доступ до пам’яті ... 108
Резюме ... 108
Вправи ... 109

Розділ 4.Компоненти комп’ютерної системи ... 111

Технічні вимоги ... 112
Підсистема пам’яті ... 112
Знайомство з польовими МОН-транзисторами ... 113
Побудова схем динамічної пам’яті за допомогою польових МОН-транзисторів ... 116
Конденсатор ... 116
Бітова комірка динамічної пам’яті ... 117
SDRAM DDR5 ... 120
DDR для графіки ... 123
Попередня вибірка ... 123
Підсистема введення-виведення ... 124
Паралельні та послідовні шини даних ... 125
PCI Express ... 127
SATA ... 129
M.2 ... 130
USB ... 130
Thunderbolt ... 131
Графічні дисплеї ... 132
VGA ... 133
DVI ... 133
HDMI ... 134
DisplayPort ... 134
Мережевий інтерфейс ... 135
Ethernet ... 135
Wi-Fi ... 137
Клавіатура і миша ... 138
Клавіатура ... 138
Миша ... 139
Технічні характеристики сучасної комп’ютерної системи ... 140
Резюме ... 142
Вправи ... 142

Розділ 5.Апаратно-програмний інтерфейс ... 143

Технічні вимоги ... 144
Драйвери пристроїв ... 144
Паралельний порт ... 145
Драйвери пристроїв PCIe ... 147
Структура драйверів пристроїв ... 148
Базова система введення-виведення (BIOS) ... 150
Єдиний розширюваний інтерфейс вбудованого ПЗ (UEFI) ... 153
Процес завантаження операційної системи ... 154
Завантаження при використанні BIOS ... 155
Завантаження під час використання UEFI ... 156
Довірене завантаження ... 157
Мобільні пристрої ... 158
Операційні системи ... 158
Процеси і потоки ... 161
Алгоритми планування та пріоритет процесу ... 164
Багатопроцесорність ... 168
Резюме ... 170
Вправи ... 170

Розділ 6.Спеціалізовані обчислення ... 171

Технічні вимоги ... 172
Обчислення в реальному часі ... 172
Операційні системи реального часу ... 174
Цифрове оброблення сигналів ... 178
АЦП і ЦАП ... 178
Особливості апаратних засобів DSP ... 181
Алгоритми обробки сигналів ... 184
Обробка даних у графічних процесорах ... 189
Графічні процесори як процесори обробки даних ... 191
Приклади специфічних архітектур ... 194
Резюме ... 196
Вправи ... 197

Розділ 7.Архітектура процесора і пам'яті ... 199

Технічні вимоги ... 199
Фон-нейманівська, гарвардська та модифікована гарвардська архітектури ... 200
Фон-нейманівська архітектура ... 200
Гарвардська архітектура ... 202
Модифікована гарвардська архітектура ... 203
Фізична та віртуальна пам'ять ... 205
Віртуальна пам'ять зі сторінковою організацією ... 208

Біти стану сторінок 213
Пупи пам'яті 214
Блок керування пам’яттю 215
Резюме 219
Вправи 219

Розділ 8.Методи підвищення продуктивності 221

Технічні вимоги 222
Кеш-пам’ять 222
Багаторівневе кешування в процесорах 224
Статична оперативна пам'ять 225
Кеш першого рівня 227
Кеш із прямим відображенням 228
Набірно-асоціативний кеш 231
Повністю асоціативний кеш 233
Політики запису в кеш процесора 234
Кеші процесора другого і третього рівнів 235
Конвеєрна обробка інструкцій 237
Суперконвеєризація 240
Конфлікти конвеєризації 242
Мікрооперації та перейменування регістрів 244
Умовне розгалуження 245
Одночасна багатопоточність 246
Модель обробки SIMD 248
Резюме 249
Вправи 250

Розділ 9.Спеціалізовані розширення процесорів 252

Технічні вимоги 252
Привілейовані режими процесора 253
Обробка переривань і винятків 253
Кільця захисту 257
Режим супервізора і режим користувача 259
Системні виклики 260
Арифметика з плаваючою комою 261
Співпроцесор 8087 для обчислень з плаваючою комою 264
Стандарт обчислень з плаваючою комою IEEE 754 266
Керування живленням 267
Динамічна зміна напруги і частоти 268
Керування безпекою системи 270
Довірений платформний модуль 272

Протидія кібератакам 274
Резюме 275
Вправи 275

Розділ 10.Сучасні архітектури і набори інструкцій процесорів 277

Технічні вимоги 277
Архітектура та набір інструкцій x86 278
Набір регістрів архітектури x86 280
Режими адресації x86 284
Категорії інструкцій x86 286
Формати інструкцій x86 293
Мова асемблера x86 293
Архітектура та набір інструкцій x64 296

Передмова

Я – розробник програмного забезпечення, а не інженер-електронник. Протягом своєї кар'єри я розробляв найрізноманітніше програмне забезпечення для вирішення безлічі різних проблем. Однак за випадковістю або примхою долі значна частина моєї кар'єри була пов'язана зі створенням програмного забезпечення, істотно ближчого до апаратних засобів, ніж у багатьох, а можливо, і в більшості, розробників програм у наші дні.

У перші роки мого захоплення комп'ютерами я швидко зрозумів, що неймовірно недосконалі за сьогоднішніми мірками пристрої, до яких у мене був доступ, не зможуть розв'язувати цікаві задачі, якщо я не навчуся програмувати їх на асемблері. Тому я навчився писати програми на асемблері 780, а потім і на асемблерах для процесорів 6502 і 80x86.

Програмування на асемблері багато в чому відрізняється від написання програм на мовах більш високого рівня. Тут ви відчуваєте, як працює обладнання. Ви не можете ігнорувати механізм розподілу пам'яті, вам необхідно підлаштувати свій код під нього. Ви не можете не взяти до уваги наявні у вашому розпорядженні регістри, це ваші змінні, і вам потрібно ретельно дотримуватися суворого порядку в керуванні їхньою роботою. Ви також вчитеся налагодження взаємодії з іншими пристроями через порти введення-виведення, адже по суті це єдиний спосіб взаємодії цифрових пристроїв один з одним.

Якось, працюючи над особливо складним завданням, я прокинувся посеред ночі і зрозумів, що мені снився асемблер 80x86.

Моя кар'єра і, що важливіше, наявне в моєму розпорядженні обладнання розвивалися. На той час я отримав роботу своєї мрії у відділі досліджень та розробок компанії-виробника комп'ютерів Моє завдання полягало у покращенні взаємодії операційних систем з нашим обладнанням, і я створював драйвери пристроїв використовувати унікальні функції наших персональних комп'ютерів. І в такій роботі також було важливо мати гарні прикладні знання у тому, як працюють апаратні засоби.

Розробка програмного забезпечення розвивалася. Мови, які ми використовували, ставали абстрактнішими, операційні системи, віртуальні машини, контейнери та загальнодоступна хмарна інфраструктура все частіше приховували

від нас, розробників програм, деталі використовуваного обладнання. Нещодавно в соціальній мережі я спілкувався з програмістом на LISP, який не розумів, що зрештою його чудові функціональні декларативні структури перетворюються на коди операцій і значення в регістрах центрального процесора. Схоже, у нього не було робочої моделі того, як працюють комп'ютери, на яку він зміг би покластися. Йому не потрібно було в цьому розбиратися, але я думаю, що якби він це зробив, то став би кращим програмістом, ніж був.

Наприкінці своєї кар'єри я працював над високопродуктивними системами світового класу. Перед колективом, який я очолював, було поставлено завдання створити одну з найефективніших фінансових бірж у світі.

Для того щоб зробити це, нам знову потрібно було ґрунтовно розібратися в тому, як працює обладнання нашої системи. Це дало нам змогу повною мірою скористатися вражаючою продуктивністю, яку пропонують сучасні апаратними засобами. У цей час, щоб спробувати описати наш підхід, ми запозичили термін з автоспорту. У 1970-х роках найкращим гонщиком «Формули-1» був Джекі Стюарт. В одному з інтерв'ю у нього запитали: «Чи потрібно володіти навичками інженера, щоб стати великим гонщиком?» Джекі відповів: «Ні, але ви повинні відчувати «механічну симпатію» до автомобіля». По суті, вам необхідно добре розуміти можливості використовуваного обладнання, щоб максимально ефективно використовувати всі його переваги.

Ми прийняли цю ідею «механічної симпатії» і застосували її в нашій роботі. Наприклад, найбільші втрати продуктивності в нашій торговій системі були пов'язані з відсутністю в кеш-пам'яті необхідних даних. Якщо дані, які ми хотіли обробити, були відсутні у відповідному кеші, коли вони були необхідні, продуктивність нашої системи падала на порядки. Тому нам потрібно було спроектувати наш код, навіть якби він був написаний мовою високого рівня і запущений на віртуальній машині, таким чином, щоб домогтися максимальної ймовірності наявності потрібних даних у кеші. Нам потрібно було зрозуміти і навчитися керувати паралелізмом у наших багатоядерних процесорах, а також знаходити і отримувати вигоду з таких аспектів, як рядки кеша процесора, переважно блокова структура пам'яті та інших пристроїв зберігання даних. Результатом стало досягнення таких рівнів продуктивності, які деякі люди вважали неможливими. Сучасне обладнання вражає своїми можливостями, коли ви використовуєте їх повною мірою.

Цей інтерес до обладнання пов'язаний не тільки з високопродуктивними обчисленнями. Незважаючи на розбіжність у конкретних оцінках, усі згодні з тим, що значна частина вуглецю, що виробляється нами як біологічним видом, зумовлена енергопостачанням центрів оброблення даних, у яких зберігається наш код. Я не можу уявити собі жодної галузі людської діяльності, яка була б настільки ж неефективною, як програмне забезпечення, для більшості систем цілком досяжне збільшення швидкості до 100 разів, якщо ви трохи попрацюєте над керуванням потоком

інформації через ваше обладнання. Майже для всіх систем можна домогтися тисячократного збільшення швидкості за певної цілеспрямованої роботи, але навіть якби ми могли отримати хоча б десятикратне прискорення за рахунок кращого розуміння того, як працює наш код і як він використовує обладнання, на якому він працює, ми могли б зменшити вуглецевий слід обчислень також у 10 разів. Ця ідея набагато важливіша, ніж підвищення продуктивності лише заради самої продуктивності.

Зрештою, є певний ступінь необхідного розуміння того, як працює ваш комп'ютер, і є ризик втратити уявлення про те, як функціонує обладнання, від якого ми всі залежимо. Зізнаюся, я фанат своєї справи. Я люблю розбиратися в тому, як усе працює. Не всім потрібно досконально знати, як працює обладнання, але ставитися до нього як до якоїсь магії – погана ідея, тому що це не магія. Це інженерна справа, а інженерна практика завжди пов'язана з компромісами. Ви будете здивовані, як часто принципи роботи вашого обладнання проявляють себе і впливають на поведінку ваших програм, якими б високорівневими вони не були, навіть якщо йдеться про написання коду для хмарних систем на LISP.

Для таких, як я, друге видання книжки Джима Леділна «Сучасна архітектура та організація комп'ютерів» – це захоплення.

Я не інженер-електронник і не хочу ним бути. Однак для мене життєво важливою частиною моїх навичок як розробника програмного забезпечення є наявність хорошої практичної моделі того, як насправді працюють апаратні засоби, на які я покладаюся. Я хочу підтримувати і розвивати концепцію «механічної симпатії».

Ця книга веде нас від основних концепцій обчислень через розгляд перших комп'ютерів і перших процесорів до потенціалу квантових обчислень і інших напрямів найближчого майбутнього, які можуть бути покладені в основу нашої обчислювальної техніки. Можливо, ви захочете зрозуміти, як працюють сучасні процесори, і досконально розібратися з їхньою вражаючою ефективністю та їхньою здатністю підтримувати свою роботу, витягуючи потрібні дані зі сховищ, які в сотні разів повільніші за них самих. Вас також можуть зацікавити складні ідеї, що виходять за межі лише апаратних засобів, наприклад про те, як працює майнінг криптовалют або який вигляд має архітектура сучасного самокерованого автомобіля. Ця книга може відповісти на ці та багато інших питань.

Я думаю, що не тільки фахівці з теорії обчислень та інженери з обчислювальної техніки, а й розробники програмного забезпечення зможуть краще виконувати свою роботу, якщо матимуть деяке уявлення про те, як працюють пристрої, які вони використовують у своїй повсякденній роботі. Намагаючись розібратися з великою і складною проблемою в програмному коді, я досі часто думаю: «Стоп! Адже насправді це лише біти, байти і коди операцій, то що ж тут відбувається?» Це схоже на те, як

хімік, розбираючись у молекулах і хімічних сполуках, може повернутися до основних принципів, щоб розв'язати будь-яке складне завдання. Це реальні першооснови, і вони можуть допомогти всім нам краще розуміти їх.

Я знаю, що буду регулярно звертатися до цієї книги протягом багатьох років, і сподіваюся, що вам сподобається робити те саме.

Дейв Фарлі (Dave Farley),
незалежний консультант з розробки програмного забезпечення
і засновник компанії Continuous Delivery Ltd

Укладачі

Про автора

Джим Ледін (Jim Ledin) – виконавчий директор компанії Ledin Engineering, Inc. Джим є експертом у сфері проєктування і тестування вбудованого програмного забезпечення та апаратних засобів, а також експертом у сфері оцінки кібербезпеки систем і тестування на проникнення. Джим має ступінь бакалавра в галузі аерокосмічної техніки від Університету штату Айова і ступінь майстра в галузі електротехніки та обчислювальної техніки від Технологічного інституту Джорджії. Він є зареєстрованим професійним інженером-електриком у штаті Каліфорнія, сертифікованим фахівцем з безпеки інформаційних систем (Certified Information System Security Professional, CISSP), сертифікованим етичним хакером (Certified Ethical Hacker, CEH) і сертифікованим фахівцем із тестування на проникнення (Certified Penetration Tester, CPT).

Я хочу подякувати моїй дружині Лінді та доньці Емілі за їхнє терпіння і підтримку в той час, коли я був зосереджений на роботі над цим проєктом. Я люблю вас, мої дорогі!

Хочу також подякувати доктору Сарі М. Нойвірт і Витоку Ієраса за їхню старанну роботу з рецензування кожного розділу цієї книги. Ваш внесок допоміг мені створити набагато кращу книгу!

Окрема подяка Дейву Фарлі за таку красномовну передмову.

Про рецензентів

Д-р Сара М. Нойвірт (Dr. Sarah M. Neuwirth) – науковий співробітник із докторським ступенем у групі модульних суперкомп’ютерів і квантових обчислень Університету Гете (Франкфурт, Німеччина). Вона також обіймає посаду запрошеного дослідника в Юліхському дослідницькому центрі суперкомп’ютерів (Німеччина). Сара має більш ніж дев’ятирічний досвід роботи в академічній сфері. У сферу її дослідницьких інтересів входять високопродуктивні системи зберігання даних, паралельне введення-виведення та файлові системи,

модульні суперкомп'ютери (розукрупнення ресурсів і віртуалізація), стандартизований порівняльний аналіз кластерів, високопродуктивні обчислення та мережі, розподілені системи та протоколи зв'язку. Крім того, Сара розробила навчальну програму і протягом останніх дев'яти років вела курси з паралельної комп'ютерної архітектури, високопродуктивних мереж із внутрішньосистемною комутацією та розподілених систем. У 2018 р. Сара отримала докторський ступінь з теорії обчислювальних машин у Гейдельберзькому університеті (Німеччина). Вона захистила науковий ступінь з відзнакою (summa cum laude) і була удостоєна наукової нагороди ZONTA 2019 за свою видатну дисертацію. Сара також має ступінь магістра наук (2012) і ступінь бакалавра наук (2010) у галузі теорії обчислень і математики Університету Мангейма (Німеччина). Вона виступала в ролі технічного рецензента для кількох престижних конференцій і журналів, присвячених високопродуктивним обчисленням, зокрема для серії конференцій IEEE/ACM SC, ACM ICPP, IEEE IDPDS, IEEE HPCC, семінару PDSW у рамках IEEE/ACM SC, семінару PERMAVOST у межах ACM HPDC, ACM TOCS, IEEE Access і Elsevier FGCS.

Істок Йерас (Iztok Jeras) здобув ступінь бакалавра електротехніки та ступінь магістра з теорії обчислювальних машин у Люблянському університеті. Він працював у кількох словенських компаніях над мікроконтролерами, рішеннями на основі програмованих логічних інтегральних схем (field-programmable gate array, FPGA) і спеціалізованих інтегральних схем (application-specific integrated circuit, ASIC), а також над вбудованим програмним забезпеченням та операційною системою Linux. У вільний час Істок досліджує клітинні автомати і бере участь у проєктах розробки цифрових систем із відкритим вихідним кодом. Останнім часом він зосередився на архітектурі набору інструкцій RISC-V.

Приєднуйтесь до нашої спільноти в Discord!

Приєднуйтесь до спільноти цієї книжки в Discord для участі в щомісячних авторських семінарах «Запитай про що завгодно»:

https://discord.gg/7h8aNRhRuY



Вступ

Вітаю читачів другого видання книги «Сучасна архітектура та організація комп'ютерів». Мені було дуже приємно отримати безліч відгуків і коментарів від читачів першого видання. Я ціную будь-які відгуки своїх читачів, особливо тих, хто звертає мою увагу на будь-які помилки або упущення.

У цій книзі представлено основні технології та компоненти, що використовуються в архітектурах сучасних процесорів і комп'ютерів, і обговорюється, як різні архітектурні рішення допомагають створювати конфігурації комп'ютерів, оптимізовані для реалізації конкретних потреб.

Сильно спрощуючи ситуацію, можна сказати, що сучасні комп'ютери – це складні пристрої. Проте, якщо застосувати для їх розгляду ієрархічний принцип, то стають зрозумілими функції кожного рівня складності. У розділах цієї книги ми торкнемося дуже багато важливих тем, і в нас буде можливість лише обмеженого ознайомлення з кожною з них. Моя мета полягає в тому, щоб надати ясне введення в кожну важливу технологію і підсистему, які ви можете знайти в сучасному обчислювальному пристрої, і пояснити їхній зв'язок з іншими компонентами системи.

Це видання містить оновлену інформацію про технології, які просунулися вперед з часу публікації першого видання, до нього також додано значний обсяг нових відомостей з кількох важливих областей, пов'язаних з архітектурою комп'ютерів. Нові глави охоплюють обчислювальні архітектури для кібербезпеки, блокчейну та майнінгу біткоїнів, а також самокерованих транспортних засобів.

Безпека обчислювальних систем завжди була важливим питанням, але недавні випадки використання серйозних вразливостей у широко поширених операційних системах і додатках призвели до значних негативних наслідків, які відчуваються в країнах по всьому світу. Ці кібератаки підкреслили необхідність включення засобів захисту від кіберзагроз як основоположного елемента архітектури обчислювальних систем.

Я не буду наводити довгий список посилань для подальшого читання. Інтернет – ваш найкращий помічник у цьому відношенні.

Якщо вам вдасться оминути метушню політичної та соціальної аргументації в Інтернеті, ви опинитеся в просторій, прохолодній і тихій бібліотеці, що містить

величезну кількість накопичених людських знань. Навчіться використовувати озширені функції вашої улюбленої пошукової системи. Крім того, зможете відрізняти якісну інформацію від думок необізнаних людей. Якщо у вас є будь-які сумніви щодо інформації, яку ви шукаєте, перевірте кілька джерел. Зверніть увагу на джерело: якщо вам потрібна інформація про процесор Intel, шукайте документацію, опубліковану Intel.

До кінця цієї книжки у вас складеться чітке уявлення про комп'ютерні архітектури, які використовуються нині в найрізноманітніших цифрових системах. Крім того, ви зможете розширити своє уявлення про відповідні тенденції в архітектурних технологіях, що розробляються в даний час, а також про деякі можливі проривні досягнення найближчого майбутнього, які можуть кардинально вплинути на розвиток архітектури обчислювальних систем.

Для кого ця книга

Ця книга призначена для розробників програмного забезпечення, студентів, які вивчають обчислювальну техніку, проектувальників систем, фахівців у галузі теорії обчислювальних машин та інженерного аналізу, а також усіх, хто хоче зрозуміти архітектуру та принципи проектування, що лежать в основі сучасних комп'ютерних систем будь-якого типу – від мініатюрних вбудованих пристроїв до смартфонів і ферм хмарних серверів розміром з великий склад. Читачі також зможуть ознайомитися з тенденціями розвитку цих технологій у найближчі роки. Загальне розуміння комп'ютерних процесорів корисне, але не обов'язкове.

Як організована ця книга

Розділ 1 «Вступ до архітектури комп'ютерів» починається зі стислої історії автоматичних обчислювальних пристроїв і описує значні технологічні досягнення, що призвели до низки проривів в обчислювальних можливостях. Потім йде обговорення закону Мура з оцінкою його застосовності в попередні десятиліття і можливих наслідків для майбутнього. Основні поняття комп'ютерної архітектури подано в контексті мікропроцесора 6502.

Розділ 2 «Цифрова логіка» презентує транзистори, що використовуються як перемикальних пристроїв, і пояснює їхнє використання в побудові логічних вентилів. Потім ми побачимо, як створюються тригери та регістри шляхом об'єднання простих вентилів. Тут вводиться поняття послідовної логіки, тобто логіки, що містить інформацію про стан. Розділ закінчується обговоренням цифрових схем із синхронізацією.

Розділ 3 «Елементи процесора» починається з концептуального опису універсального процесора. Ми розглянемо концепції набору інструкцій, набору регістрів, а також

завантаження, декодування, виконання та визначення послідовності інструкцій.

Також тут обговорюються операції завантаження і збереження даних у пам'яті. Розділ містить опис інструкцій розгалуження та їх використання в циклах і умовній обробці. Наведено деякі практичні міркування, що призводять до необхідності обробки переривань і операцій введення-виведення.

У розділі 4 «Компоненти комп'ютерної системи» обговорюються пам'ять комп'ютера та її інтерфейс із процесором, у тому числі багаторівневе кешування. Описано вимоги до введення-виведення, включно з обробкою переривань, буферизацією і виділеними процесорами введення-виведення. Ми обговоримо деякі особливі вимоги до пристроїв введення-виведення, включно з клавіатурою і мишею, дисплеєм і мережевим інтерфейсом. Розділ закінчується описовими прикладами цих компонентів у сучасних комп'ютерних системах, серед яких мобільні пристрої, персональні комп'ютери, ігрові системи, хмарні сервери та спеціалізовані системи машинного навчання.

У розділі 5 «Апаратно-програмний інтерфейс» обговорюється реалізація високорівневих функцій, які має надавати операційна система комп'ютера, включно з введенням виведенням для накопичувачів, обміном даними мережами та взаємодією з користувачами. У цьому розділі описуються рівні програмного забезпечення, що реалізують ці функції, починаючи з рівня набору інструкцій і регістрів процесора. Також описано функції операційної системи, зокрема її завантаження, багатопроцесорність і багатопоточність.

Розділ 6 «Спеціалізовані обчислення» досліджує області обчислень, які, як правило, менш очевидні для більшості користувачів, включно із системами реального часу, цифровою обробкою сигналів і обробкою даних у графічних процесорах. Ми обговоримо унікальні вимоги, пов'язані з кожною з з цих областей, і розглянемо приклади сучасних пристроїв, що реалізують ці функції.

Розділ 7 «Архітектура процесора і пам'яті» докладно розглядає архітектурні рішення сучасних процесорів, включаючи фон-нейманівську, гарвардську і модифіковану гарвардську. Тут обговорюється реалізація віртуальної пам'яті зі сторінковою організацією. Представлено практичну реалізацію функцій керування пам'яттю в архітектурі комп'ютера, а також функцій блоку керування пам'яттю.

У розділі 8 «Методи підвищення продуктивності» обговорюється низка методів підвищення продуктивності, які зазвичай використовуються для досягнення максимальної швидкості виконання програм у реальних комп'ютерних системах. У цьому розділі розглядаються найважливіші методи підвищення продуктивності систем, включно з використанням кеш-пам'яті, конвеєрного опрацювання інструкцій, паралелізму інструкцій і опрацюванням на основі принципу «одна інструкція, безліч даних» (single instruction-multiple data SIMD).

Розділ 9 «Спеціалізовані розширення процесорів» фокусується на розширеннях, які зазвичай реалізують на рівні набору інструкцій процесора, з метою надання додаткових системних можливостей крім загальних вимог до обробки даних. Представлені розширення містять привілейовані режими процесора, арифметику з плаваючою комою, керування живленням і безпекою системи.

Глава 10 «Сучасні архітектури та набори інструкцій процесорів» досліджує особливості архітектури та наборів інструкцій сучасних процесорів, включно з x86, x64 і ARM. Однією з проблем, що виникають під час виробництва сімейства процесорів протягом кількох десятиліть, є необхідність підтримувати зворотну сумісність із кодом, написаним для процесорів попередніх поколінь. Необхідність підтримки успадкованого коду призводить до збільшення складності процесорів пізніших поколінь. У цьому розділі розглянуто деякі атрибути цих процесорних архітектур, які обумовлені підтримкою застарілих вимог.

Розділ 11 «Архітектура та набір інструкцій RISC-V» представляє захоплюючу нову архітектуру процесорів RISC-V (вимовляється як «ризик файв») і відповідний набір інструкцій. RISC-V – це вільно поширювана специфікація з повністю відкритим вихідним кодом для обчислювальної архітектури зі скороченим набором інструкцій. Випущено повну специфікацію набору інструкцій, і наразі доступно кілька апаратних реалізацій цієї архітектури. Триває розроблення специфікацій для низки розширень набору інструкцій. У цьому розділі розглядаються функції, доступні в архітектурі RISC-V, варіанти її реалізації та представлено набір інструкцій RISC-V. Ми також обговоримо застосування архітектури RISC-V у мобільних пристроях, персональних комп’ютерах і серверах.

Розділ 12 «Віртуалізація процесорів» представляє концепції, пов’язані з віртуалізацією процесорів, і пояснює численні переваги, які виникають у результаті застосування віртуалізації. Розділ містить приклади віртуалізації на основі інструментів та операційних систем з відкритим вихідним кодом. Ці інструменти дають змогу створювати на звичайному комп’ютері точні з погляду набору інструкцій уявлення різних комп’ютерних архітектур і операційних систем. Ми також обговоримо переваги віртуалізації при розробці та розгортанні реальних програмних додатків.

Розділ 13 «Спеціалізовані комп’ютерні архітектури» об’єднує теми, розглянуті в попередніх розділах, з метою вироблення підходу до створення архітектури комп’ютерної системи для виконання унікальних вимог користувача. Ми обговоримо конкретні категорії сфер застосування, включно з мобільними пристроями, персональними комп’ютерами, ігровими системами, пошуковими системами і нейронні мережі.

Розділ 14 «Архітектури для забезпечення кібербезпеки та конфіденційності обчислень» зосереджено на потребах у захисті критично важливих сфер, включно із

системами національної безпеки та обробкою фінансових транзакцій. Ці системи мають бути стійкими до широкого спектра загроз кібербезпеки, зокрема до проникнення шкідливого коду, атак через приховані канали та шляхом фізичного впливу. приховані канали та шляхом фізичного доступу до апаратних засобів комп'ютерів. Серед тем, що розглядаються в цьому розділі, – загрози кібербезпеки, шифрування, цифрові підписи та архітектурні рішення для захисту апаратних засобів і програмного забезпечення. Вибух інтересу до криптовалют і їхнє зростаюче визнання основними фінансовими інститутами та торговельними підприємствами демонструють, що дана галузь обчислювальної техніки перебуває на шляху безперервного зростання. У це видання книги додано розділ про блокчейн і обчислювальні вимоги майнінгу біткоїнів.

Розділ 15 «Архітектури блокчейна і майнінгу біткоїнів» представляє концепції, пов'язані з блокчейном, відкритим, криптографічно захищеним реєстром, що містить послідовність транзакцій. За вступом слідує огляд процесу майнінгу біткоїнів, який додає транзакції в блокчейн Bitcoin і винагороджує тих, хто виконує це завдання, оплатою в біткоїнах. Для обробки біткоїнів потрібне високопродуктивне обчислювальне обладнання, яке представлено з точки зору комп'ютерної архітектури для майнінгу біткоїнів поточного покоління.

Тривале зростання числа автомобілів з можливістю часткового або повного автономного водіння вимагає надійних обчислювальних систем з високою продуктивністю, що відповідають вимогам безпечної автономної експлуатації транспортних засобів на дорогах загального користування.

Розділ 16 «Архітектури для самокерованих автомобілів» представляє можливості, які мають бути реалізовані в обчислювальних архітектурах самокерованих автомобілів. Розділ починається з обговорення вимог щодо забезпечення безпеки самокерованого автомобіля та його пасажирів, а також інших транспортних засобів, пішоходів і стаціонарних об'єктів. Потім ми розглянемо типи датчиків і джерела інформації, які надають самокерованим автомобілям вхідні дані під час руху, а також типи обчислень, необхідних для ефективного керування автомобілем. Розділ завершиться оглядом прикладу архітектури комп'ютера для самокерованого автомобіля.

У розділі 17 «Квантові обчислення та інші перспективні напрямки в обчислювальних архітектурах» оцінюються перспективи розвитку комп'ютерних архітектур. У ній розглядаються важливі досягнення та сучасні тенденції, що призвели до поточного стану комп'ютерних архітектур, а також прогнозується подальший розвиток цих тенденцій для визначення можливих майбутніх технологічних напрямків. Також обговорюються потенційно проривні технології, які можуть змінити напрямок розвитку комп'ютерних архітектур у майбутньому. На закінчення я дам рекомендації щодо деяких підходів до професійного вдосконалення архітектора комп'ютерів, які можуть допомогти сформувати набір корисних для майбутнього навичок.

Як і інші глави, кожна з трьох нових глав містить вправи, які допомагають краще зрозуміти тему глави і закріпити отриману інформацію у вашій базі знань.

Я сподіваюся, що ви отримаєте задоволення від читання оновленого видання, так само, як і я, коли працював над цією книгою. Приємного читання!

Як отримати максимальну віддачу від цієї книги

Кожен розділ цієї книги завершується набором вправ. Для того щоб отримати з книги максимальну користь і закріпити в розумі деякі з найбільш складних концепцій, я рекомендую вам попрацювати над кожною вправою. Повні рішення для всіх вправ наведені в книзі та доступні в Інтернеті за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-andOrganization-Second-Edition.**

У разі необхідності внесення змін до прикладів коду та відповідей до вправ ці зміни з'являться в цьому ж репозиторії GitHub.

Завантажте файли з прикладами коду

Комплект вихідного коду для цієї книги розміщено на GitHub за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-andOrganization-Second-Edition.** У нас також є інші комплекти вихідного коду з нашого великого каталогу книг і відео, доступних за адресою **https://github.com/PacktPublishing/**. Рекомендуємо ознайомитися з ними!

Завантажте кольорові зображення

Ми також надаємо PDF-файл із кольоровими зображеннями знімків екрана та діаграм, використаних у цій книзі. Ви можете завантажити цей файл за адресою: **https://static.packt-cdn.com/downloads/9781803234519_ColorImages.pdf** [1].

Умовні позначення та угоди

У книзі використовуються такі типографські умовні позначення.

CodelnText вказує на приклади коду в тексті, імена таблиць баз даних, імена папок, імена файлів, розширення файлів, шляхи, користувацьке введення та імена облікових записів у Twitter. Приклад: «Віднімання з використанням інструкції SBC може трохи збити з пантелику програмістів-початківців мовою асемблера 6502».

Лістинг коду позначається так:

```
; Додавання чотирьох байтів з використанням режиму
безпосередньої адресації
LDA #$04
CLC
ADC #$03
ADC #$02
ADC #$01
```

Будь-які приклади введення або виведення з використанням командного рядка записуються таким чином:

```
C:\>bcdedit
Windows Boot Manager
--------------------
identifier {bootmgr}
```

Напівжирний шрифт позначає новий термін або важливе слово, інтернет-адресу та адресу електронної пошти; у такий самий спосіб виділяються в тексті слова, які ви бачите на екрані, – назви пунктів меню або елементів діалогових вікон.
Приклад: «Оскільки тепер існують чотири набори, поле **Набір** у фізичній адресі зменшується до двох бітів, а поле **Тег** збільшується до 24 біт».



Попередження або важливі примітки мають такий вигляд.

Зв’яжіться з нами

Зворотний зв’язок з боку наших читачів завжди вітається.
Загальні відгуки. Відгуки загального характеру надсилайте на електронну пошту **feedback@packtpub.com,** вказавши в темі повідомлення назву книги. Якщо у вас є запитання щодо будь-якого аспекту цієї книги, будь ласка, напишіть нам за адресою **questions@ packtpub.com.**
Помилки. Ми доклали всіх розумних зусиль, щоб забезпечити точність вмісту книги, але помилки іноді трапляються. Якщо ви знайшли помилку в цій книзі, ми будемо вдячні, якщо ви повідомите нам про це. Зайдіть на сайт **http://www.packtpub.com/submit-errata**, виберіть свою книгу, клацніть за посиланням **Errata Submission Form** (Форма повідомлення про помилки) і введіть відомості про помилку.

Піратство. Якщо в Інтернеті вам зустрінуться незаконні копії наших творів у будь-якій формі, ми будемо вдячні, якщо ви повідомите нам адресу їхнього місцезнаходження або назву веб-сайту. Будь ласка, надішліть нам посилання на такий матеріал за адресою **copyright@packtpub.com.**

Якщо ви хочете стати автором. Якщо є тема, у якій ви маєте досвід, і ви зацікавлені в написанні або участі в написанні книги, будь ласка, відвідайте веб-сайт **http://authors.packtpub.com.**

Поділіться своїми думками

Ми будемо раді дізнатися вашу думку після того, як ви прочитаєте друге видання книги «Сучасна архітектура та організація комп'ютерів». Відскануйте наведений нижче QR-код, щоб перейти на сторінку рецензування цієї книги та поділитися своєю думкою.

Ваш відгук важливий для нас і технічної спільноти, він допоможе нам переконатися, що ми надаємо інформацію найвищої якості.

1

Введення в архітектуру комп'ютерів

Архітектура автоматичних обчислювальних систем пройшла довгий шлях від перших механічних обчислювачів, створених майже два століття тому, до широкого спектра сучасних електронних комп'ютерних технологій, які ми прямо або побічно використовуємо щодня. На цьому шляху були періоди поступових технологічних поліпшень, що чергувалися з проривними досягненнями, які різко змінювали траєкторію розвитку галузі. Можна очікувати, що ці тенденції збережуться і в найближчі роки.

У 1980-х роках, на зорі розвитку персональних обчислювальних пристроїв, студентам і технічним фахівцям, які прагнули дізнатися якомога більше про комп'ютерні технології, був доступний лише обмежений набір предметів для вивчення. Якщо вони мали власний комп'ютер, то це, найімовірніше, був IBM PC або Apple II. Якщо вони працювали в організації з обчислювальним центром, то могли б використовувати велику ЕОМ від IBM або міні-комп'ютер VAX від Digital Equipment Corporation. Ці приклади та обмежене число подібних систем охоплювали практично всі можливості знайомства більшості людей з комп'ютерними системами того часу.

Сьогодні існує безліч спеціалізованих обчислювальних архітектур для задоволення найрізноманітніших потреб користувачів для задоволення найрізноманітніших потреб користувачів. У своїх кишенях ми носимо мініатюрні комп'ютери, які можуть здійснювати телефонні виклики, записувати відео і надавати повнофункціональний доступ в Інтернет. Персональні комп'ютери (ПК) залишаються популярними у форматі, зовні схожому на ПК минулих десятиліть. Однак сучасні ПК на кілька порядків перевершують попередні покоління за обчислювальною потужністю, обсягом пам'яті, дисковим простором, графічною продуктивністю та доступними способами зв'язку. Ці можливості дають змогу сучасним ПК легко розв'язувати завдання, виконання яких було б немислимим на ранніх ПК, як-от формування тривимірних зображень із високою роздільною здатністю в реальному часі.

Компанії, що пропонують веб-сервіси сотням мільйонів користувачів, створюють величезні сховища, заповнені тисячами тісно скоординованих комп’ютерних систем, здатних реагувати на постійний потік призначених для користувача запитів з надзвичайною швидкістю і точністю. Системи машинного навчання навчаються на основі аналізу величезних обсягів даних для виконання складних дій, таких як керування автомобілями.

Цей розділ починається з представлення деяких обчислювальних пристроїв, які відіграли важливу роль в історії галузі, і пов’язаних з ними технологічних проривів. Потім ми розглянемо деякі важливі сучасні тенденції, пов’язані з технологічними досягненнями, і познайомимося з основними концепціями комп’ютерної архітектури, включно з уважним вивченням мікропроцесора 6502 і його набору інструкцій. У цьому розділі буде розглянуто такі теми:

- еволюція автоматичних обчислювальних пристроїв;
- закон Мура;
- архітектура комп’ютерів.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-andOrganization-Second-Edition.**

Еволюція автоматичних обчислювальних пристроїв

У цьому розділі розглядаються деякі класичні машини з історії автоматичних обчислювальних пристроїв, при цьому основна увага приділяється важливим досягненням, втіленим у кожній з них. Аналітичну машину Беббіджа включено сюди завдяки безлічі геніальних нововведень, представлених у її конструкції. Обговорення інших систем зумовлене тим, що вони втілювали в собі значні технологічні здобутки та за час свого існування виконали значну практичну роботу.

Аналітична машина Чарльза Беббіджа

Незважаючи на те що робоча модель цієї **аналітичної машини** ніколи не була побудована, у докладних записах, які вів **Чарльз Беббідж** із 1834 р. до своєї смерті в 1871 р., було описано обчислювальну архітектуру, що здавалася цілком завершеною та працездатною. Ця аналітична машина призначалася для використання як універсальний програмований обчислювальний пристрій. Її конструкція була повністю механічною, а виготовити машину передбачалося здебільшого з латуні. Аналітична машина мала приводитися в рух валом від парового двигуна.

Для своєї аналітичної машини Беббідж запозичив перфокарти ткацького верстата Жаккара, циліндри, що обертаються, зі штифтами, які використовують у музичних скриньках, і технології своєї більш ранньої різницевої машини (також не закінченої за його життя, що була скоріше спеціалізованим лічильним пристроєм, ніж комп'ютером), у решті її конструкція була оригінальним творінням Беббіджа.

На відміну від більшості сучасних комп'ютерів, ця аналітична машина представляла числа в десятковому вигляді зі знаком. Рішення використовувати числа з основою 10, а не логіку більшості сучасних комп'ютерів з основою 2, було зумовлене фундаментальною відмінністю між механічними технологіями і цифровою електронікою. Виготовлення механічних коліс із 10 положеннями – нескладне завдання, тому Беббідж вибрав звичний людині формат із основою 10, оскільки він був ненабагато складнішим із технічної точки зору, ніж використання будь-якої іншої системи числення. З іншого боку, прості цифрові схеми не здатні підтримувати 10 різних станів з тією ж легкістю, з якою це досягається за допомогою механічного колеса.

Усі числа в аналітичній машині складалися з 40 десяткових цифр. Велика кількість цифр, імовірно, була обрана для полегшення вирішення проблем із числовим переповненням. Ця аналітична машина не підтримувала математику з плаваючою комою.

Кожне число зберігалося на вертикальній осі, що містила 40 коліс, кожне з яких могло перебувати в 10 положеннях, що відповідають цифрам від 0 до 9. Сорок перше колесо визначало знак: будь-яке парне число на цьому колесі представляло позитивний знак, а будь-яке непарне число – негативний. Вісь аналітичної машини була чимось аналогічна регістру, використовуваному в сучасних процесорах, за винятком того, що зчитування числа вісі було руйнівним – після цієї операції число оберталося на 0. Якщо потрібно було зберегти значення осі після зчитування, інша вісь повинна була зберегти копію значення під час зчитування. Числа переносили з однієї осі на іншу або використовували в обчисленнях шляхом зачеплення шестерні з кожним колесом і обертання колеса для вилучення числового значення. Набір осей, що слугують системною пам'яттю, називався *складом*.

Для додавання двох чисел використовувався процес, що нагадує метод додавання, якого навчають школярів. Припустимо, що число, що зберігається на одній осі, назвемо його доданком, має бути додано до числа на іншій осі, яке ми назвемо акумулятором. Машина повинна була з'єднати колесо кожної цифри доданка з відповідним колесом цифри акумулятора за допомогою зубчастої передачі. Потім вона повинна була одночасно обертати колесо кожної цифри доданка в напрямку нуля, тоді як колесо цифри акумулятора здійснювало еквівалентний поворот у зростаючому напрямку. Якщо цифра акумулятора здійснювала перехід від 9 до 0, наступна за старшинством цифра акумулятора збільшувалася на 1. Ця операція перенесення могла

поширюватися на стільки цифр, скільки було необхідно (подумайте про додавання 1 до 999 999 999). До кінця процесу вісь доданка мала містити значення 0, а вісь акумулятора – суму двох чисел. Поширення переносів від однієї цифри до іншої було найскладнішою з погляду механіки частиною процесу додавання.

Операції в аналітичній машині виконували за допомогою циліндрів, що обертаються, схожих на барабан музичної скриньки, у конструкції, що називається млином, яка аналогічна блоку керування сучасного процесора.

Кожна інструкція аналітичної машини була закодована у вертикальному ряді позицій на циліндрі, де наявність або відсутність штифта в певному місці або задіювала частину механізму машини, або залишала стан відповідної секції незмінним. Згідно з оцінкою швидкості виконання операцій, зробленою Беббіджем, додавання двох 40-значних чисел, включно з поширенням переносів, мало зайняти близько 3 секунд.

Беббідж розробив для своєї машини кілька важливих концепцій, які залишаються актуальними і для сучасних комп’ютерних систем. Його конструкція підтримувала певний ступінь паралельного опрацювання, що передбачало одночасне виконання операцій множення і додавання, що прискорювало обчислення серій значень, призначених для виведення у вигляді числових таблиць. Математичні операції, такі як додавання, підтримували деяку форму конвеєрної обробки, за якої послідовні операції над різними значеннями перекриваються в часі.

Беббідж добре знав про труднощі, пов’язані зі складними механічними пристроями, як-от тертя, люфт шестерень і знос із плином часу. Для того щоб запобігти помилкам, спричиненим цими ефектами, у машину було вбудовано механізми, звані блокуваннями, які застосовувалися під час передачі даних між осями. Блокування змушували колеса цифр займати правильні положення і запобігали накопиченню невеликих помилок, викликаних поступовим зміщенням коліс у напрямку неправильних значень. Використання блокувань аналогічне посиленню потенційно слабких вхідних сигналів для отримання сильніших вихідних сигналів за допомогою цифрових логічних вентилів у сучасних процесорах.

Для програмування аналітичної машини передбачалося використовувати перфокарти. Крім того, в ній мали підтримуватися операції розгалуження і вкладені цикли. Найскладніша програма, призначена для виконання на цій аналітичній машині, була розроблена Адою Лавлейс для обчислення чисел Бернуллі – важливої послідовності в теорії чисел. Код машини для виконання цих обчислень визнано першою опублікованою комп’ютерною програмою значної складності. Беббідж побудував пробну модель частини млина своєї аналітичної машини, яка нині виставлена в Музеї науки в Лондоні.

ENIAC

Створення машини **ENIAC**, назва якої розшифровується як **Electronic Numerical Integrator and Computer** (електронна лічильна машина-інтегратор), було завершене 1945 року, і вона стала першим програмованим універсальним електронним комп'ютером. Система споживала 150 кВт електроенергії, займала 170 квадратних метрів площі і важила 27 тон.

Її конструкція була заснована на електронних лампах, діодах і реле. Машина ENIAC містила понад 17 000 електронних ламп, які виконували функції перемикальних елементів.

Подібно до аналітичної машини Беббіджа, вона використовувала десяткове представлення 10-значних десяткових чисел, реалізоване за допомогою 10-позиційних кільцевих лічильників (кільцевий лічильник буде обговорено в розділі 2).

Вхідні дані вводилися за допомогою пристрою читання перфокарт від IBM, а результати обчислень виводилися за допомогою перфораційної машини.

Архітектура ENIAC була здатна виконувати складні послідовності кроків обробки, включно з циклами, розгалуженням і підпрограмами. Система містила 20 десятирозрядних акумуляторів, які діяли як регістри сучасних комп'ютерів. Спочатку у машини не було ніякої пам'яті, крім акумуляторів. Якщо проміжні значення були потрібні для використання в подальших обчисленнях, ці дані доводилося записувати на перфокарти і в міру необхідності зчитувати назад у машину. ENIAC могла виконувати близько 385 операцій множення за секунду.

Програми ENIAC являли собою дротові з'єднання на комутаційній панелі і таблиці функцій на основі перемикачів. Програмування системи було трудомістким процесом, на виконання якого у групи талановитих жінок-програмістів часто йшли тижні. Надійність була проблемою, оскільки вакуумні лампи регулярно виходили з ладу, що вимагало щоденного виконання процедури пошуку та усунення неполадок.

У 1948 р. ENIAC було поліпшено за рахунок додавання можливості програмування системи за допомогою перфокарт замість комутаційних панелей. Це значно прискорило розробку програм. Виступаючи консультантом з цього оновлення, Джон фон Нейман запропонував архітектуру обробки, засновану на єдиній ділянці пам'яті, що містить програмні інструкції та дані, блоці обробки з арифметико-логічним пристроєм та регістрами, а також блоці керування, який містить регістр інструкцій і програмний лічильник. Багато сучасних процесорів успадковують цю загальну структуру, відому тепер як **архітектура фон Неймана**. Ми детально обговоримо її в розділі 3.

Ранні сфери застосування машини ENIAC охоплювали аналіз, пов'язаний із розробленням водневої бомби, і розрахунок таблиць стрільби для далекобійної артилерії.

IBM PC

За роки, що минули після створення ENIAC, відбулося кілька технологічних проривів, які призвели до значного прогресу в комп’ютерній архітектурі.

- Поява транзистора, який винайшли в 1947 р. Джон Бардін, Волтер Браттейн і Вільям Шоклі, забезпечила значне поліпшення порівняно з технологією електронних ламп, що переважала в той час. Транзистори були швидшими, компактнішими, споживали менше енергії, а після оптимізації виробничих процесів стали набагато надійнішими за лампи, що зазнавали відмов.
- Після появи в 1958 р. ідеї інтегральних схем, запропонованої Джеком Кілбі з Texas Instruments, почався процес об’єднання великої кількості раніше окремих компонентів на одному кремнієвому чипі.
- У 1971 р. компанія Intel почала виробництво першого мікропроцесора Intel 4004, який надійшов у відкритий продаж. Модель 4004 призначалася для використання в електронних калькуляторах і була спеціалізована для роботи з 4-бітними двійково-десятковими числами.

Зі скромного початку у вигляді Intel 4004 мікропроцесорна технологія швидко розвивалася протягом наступного десятиліття: кількість елементів схем на кожному чіпі стрімко зростала, а можливості мікропроцесорів, реалізованих на цих чіпах, розширювалися.

Мікропроцесор Intel 8088

IBM випустила персональний комп’ютер IBM PC 1981 р. Оригінальний ПК містив мікропроцесор Intel 8088, що працював на тактовій частоті 4,77 МГц, і мав 16 Кбайт **оперативної пам’яті** (оперативний запам’ятовувальний пристрій, ОЗП) з можливістю розширення до 256 Кбайт. Він комплектувався одним або (за додатковим запитом) двома дисководами для гнучких дисків. Також був доступний кольоровий монітор. Пізніші версії цього ПК підтримували більше пам’яті, але оскільки частина адресного простору була зарезервована для відеопам’яті та **постійної пам’яті** (постійний запам’ятовувальний пристрій, ПЗП), ця архітектура могла підтримувати ОЗП максимальним обсягом 640 Кбайт.

Процесор 8088 містив чотирнадцять 16-розрядних регістрів. Чотири з них виконували функції регістрів загального призначення (ax, bx, cx і dx). Чотири інших були регістрами сегментів пам’яті (cs, DS, SS і ES), які розширювали адресний простір до 20 біт. Сегментна адресація функціонувала шляхом додавання 16-бітного значення сегментного регістра, зрушеного вліво на 4 біти, до 16-бітного значення зсуву, що міститься в інструкції, для отримання адреси фізичної пам’яті в діапазоні до 1 Мбайт.

Іншими регістрами процесора 8088 були **покажчик стека** *(sp)*, **покажчик бази** *(BP)*, **індекс джерела** *(SI)*, **індекс приймача** *(di)*, **покажчик інструкцій** *(IP)* і **прапори**

стану *(FLAGS)*. У сучасних процесорах сімейства x86 реалізовано архітектуру, напрочуд схожу на цей набір регістрів (деталі архітектури x86 описано в розділі 10). Найочевидніші відмінності між процесорами 8088 і x86 – збільшення розрядності регістрів у x86 до 32 бітів і додавання пари сегментних регістрів (FS і GS), які сьогодні використовуються переважно як покажчики даних у багатопотокових операційних системах.

Процесор 8088 мав 8-розрядну зовнішню шину даних – це означало, що для читання або запису 16-бітного значення були потрібні два цикли шини. Це був крок назад за продуктивністю порівняно з більш раннім процесором 8086, який оснащувався 16-розрядною зовнішньою шиною. Однак застосування 8-розрядної шини дало змогу заощадити на виробництві ПК і забезпечило сумісність із дешевшими 8-розрядними периферійними пристроями. Такий економічний підхід до проектування допоміг знизити ціну ПК до рівня, доступного більшій кількості потенційних покупців.

Для пам’яті програм і пам’яті даних спільно використовували один і той самий адресний простір, а доступ до пам’яті 8088 здійснювали по одній шині. Інакше кажучи, у 8088 було реалізовано архітектуру фон Неймана. Набір інструкцій 8088 містив інструкції для переміщення даних, арифметичних і логічних операцій, маніпулювання рядками, передавання керування (умовні та безумовні переходи, а також виклик підпрограм і повернення з них), операцій введення-виведення і різних додаткових функцій. Для виконання однієї інструкції процесору було потрібно в середньому близько 15 тактів, що забезпечувало швидкість виконання близько 0,3 **млн інструкцій на секунду** (million instructions per second, MIPS).

Процесор 8088 підтримував дев’ять різних режимів адресації пам’яті. Така різноманітність режимів була необхідна для ефективного доступу до одного елемента за раз, а також для перебору послідовностей даних.

Сегментні регістри в архітектурі 8088 надавали, здавалося б, розумний спосіб розширення діапазону адресованої пам’яті без збільшення довжини більшості інструкцій, що посилаються на комірки пам’яті. Кожен сегментний регістр забезпечував доступ до блоку пам’яті розміром 64 Кбайт, що починається з адреси фізичної пам’яті, кратної 16 байтам. Інакше кажучи, 16-розрядний сегментний регістр представляв 20-бітну базову адресу з чотирма молодшими бітами, що дорівнюють нулю. Таким чином, інструкції могли посилатися на будь-яку комірку в сегменті завдовжки 64 Кбайт, використовуючи 16-бітове зміщення від адреси, визначеної регістром сегмента.

Регістр SS задавав місце розташування сегмента стека, яке використовувалося для виділення локальної пам’яті всередині підпрограм і для зберігання адрес повернення з підпрограм.

Програми, яким було потрібно менше 64 Кбайт у кожному із сегментів коду, даних і стека, могли повністю ігнорувати сегментні регістри, оскільки ці регістри могли бути

встановлені один раз під час запуску програми (компілятори мов програмування роблять це автоматично) і залишатися незмінними під час її виконання. Дуже просто!

Усе ставало трохи складнішим, коли розмір даних програми перевищував 64 Кбайт. Застосування сегментних регістрів дало змогу створити струнке апаратне рішення, але їхнє використання доставляло багато проблем розробникам програмного забезпечення. Компілятори для архітектури 8088 розрізняли покажчики типу near і far. Покажчик типу near являв собою 16-бітове зміщення від базової адреси поточного сегментного регістра. Покажчик типу far містив 32 біти адресної інформації: 16-бітове значення регістра сегмента і 16-бітове зміщення. Покажчики типуfar споживали додаткові 16 біт пам’яті даних, а також вимагали додаткового часу на оброблення.

Для однієї операції доступу до пам’яті з використанням покажчика типу far потрібно було виконати такі кроки:

1. Зберегти поточний вміст сегментного регістра в тимчасовій комірці пам’яті.
2. Завантажити нове значення сегмента в регістр.
3. Виконати операцію доступу до даних (читання або запис, у міру необхідності) з використанням зміщення від базової адреси сегмента.
4. Відновити вихідне значення сегментного регістра.

При використанні покажчиків типу far можна було оголошувати об’єкти даних (наприклад, масив символів, що представляють документ у текстовому редакторі) розміром до 64 Кбайт. Якщо була потрібна більша структура, потрібно було придумати, як розбити її на фрагменти розміром не більше ніж 64 Кбайт і керувати ними самостійно. Унаслідок таких маніпуляцій із сегментним регістром програми, які потребували широкого доступу до елементів даних розміром понад 64 Кбайти, ставали досить складними, що призводило до збільшення розміру коду і уповільнення його виконання.

На материнській платі IBM PC розташовувався роз’єм для пропонованого окремо співпроцесора Intel 8087 для обчислень з плаваючою комою. Розробники 8087 винайшли формати даних і правила опрацювання для 32- і 64-розрядних чисел із плаваючою комою, які були закріплені в 1985 р. у вигляді стандарту IEEE 754 для обчислень із плаваючою комою – цей стандарт знаходить практично універсальне застосування і сьогодні. Співпроцесор 8087 може виконувати близько 50 000 операцій із плаваючою комою за секунду. Ми детально розглянемо їх у *розділі 9*.

Мікропроцесори Intel 80286 і 80386

Друге покоління комп’ютерів IBM PC – PC AT – було випущене в 1984 р. **AT** означало **Advanced Technology** (поліпшена технологія) і вказувало на деякі істотні поліпшення порівняно з оригінальним ПК, які в основному були результатом використання процесора Intel 80286. Як і 8088, 80286 був 16-розрядним процесором, який підтримував зворотну сумісність із 8088: код 8088 можна було виконувати на 80286

без змін. 80286 мав 16-розрядну шину передачі даних і 24 адресні рядки, що підтримують адресний простір розміром 16 Мбайт. 16-розрядна зовнішня шина даних підвищувала швидкість доступу до даних порівняно з 8-розрядною шиною процесора 8088. Швидкість виконання інструкцій (за такт процесора) була приблизно вдвічі вищою, ніж у 8088, у багатьох додатках. Це означало, що за тієї самої тактової частоти 80286 міг працювати вдвічі швидше, ніж 8088. Процесор оригінального PC AT мав тактову частоту 6 МГц, а його пізніша версія працювала на частоті 8 МГц. Швидкість виконання інструкцій на 80286 з тактовою частотою 6 МГц досягала приблизно 0,9 млн інструкцій на секунду, що було приблизно втричі вище, ніж у 8088.

У 80286 було реалізовано режим захищеної віртуальної адресації, призначений для підтримки багатокористувацьких операційних систем і багатозадачності.

У захищеному режимі процесор забезпечував захист пам'яті, щоб гарантувати, що програми одного користувача не вплинуть на роботу операційної системи або програм інших користувачів. Це революційне технологічне досягнення в галузі персональних комп'ютерів залишалося мало використовуваним протягом багатьох років головним чином через непомірно високу вартість додавання в комп'ютерну систему достатнього обсягу пам'яті, щоб зробити її корисною в багатокористувацькому, багатозадачному контексті.

Після 80286 наступне покоління лінійки процесорів x86 було представлено моделлю 80386, випущеною 1985 р. 80386 був 32-розрядним процесором із підтримкою 32-розрядної лінійної моделі пам'яті в захищеному режимі. Лінійна модель давала змогу програмістам безпосередньо звертатися до 4 Гбайт пам'яті, не вимагаючи маніпуляцій із сегментними регістрами. У 1986 р. компанія Compaq представила сумісний з IBM PC персональний комп'ютер на базі 80386, названий Deskpro. DeskPro постачається з версією ОС Microsoft Windows, орієнтованою на архітектуру 80386.

Процесор 80386 значною мірою підтримував зворотну сумісність із процесорами 80286 і 8088. Архітектура, реалізована в 80386, залишається поточною стандартною архітектурою сімейства x86. Ми детально обговоримо її в *розділі 10*.

Початкова версія 80386 мала тактову частоту 33 МГц, а швидкість виконання досягала приблизно 11,4 млн інструкцій на секунду. Сучасні реалізації архітектури x86 працюють у кілька сотень разів швидше, ніж початкова версія, завдяки вищій тактовій частоті, підвищенню продуктивності, зокрема широкому використанню багаторівневої кеш-пам'яті, і ефективнішому виконанню інструкцій на апаратному рівні. Ми розглянемо переваги кешпам'яті в *розділі 8*.

iPhone

У 2007 р. Стів Джобс представив світові iPhone – людство тоді й гадки не мало про те, яку користь можна було отримати з такого пристрою. iPhone був

створений на основі попередніх революційних досягнень Apple Computer, включно з комп’ютером Macintosh, випущеним у 1984 р., і музичним плеєром iPod, що з’явився в 2001 р. iPhone поєднував у собі функції iPod, мобільного телефону і комп’ютера з виходом в Інтернет.

В iPhone було скасовано апаратну клавіатуру, яка була поширена на смартфонах того часу, її замінив сенсорний екран, здатний відображати екранну клавіатуру або будь-який інший тип користувацького інтерфейсу. Крім дотиків для вибору символів на клавіатурі і натискання кнопок, екран підтримував жести кількома пальцями, які давали змогу виконувати такі дії, як масштабування фотографій.

iPhone працював під керуванням операційної системи OS X; ту саму ОС використовували на флагманських комп’ютерах Macintosh того часу.

Таке рішення одразу ж дало змогу iPhone підтримувати широкий спектр застосунків, уже розроблених для Mac, а розробники ПЗ отримали змогу швидко впроваджувати нові додатки, адаптовані для iPhone, після того як Apple дозволила сторонню розробку застосунків.

iPhone 1 був оснащений 3,5-дюймовим екраном з роздільною здатністю 320 x 480 пікселів. Він мав товщину 11,6 мм (тонше, ніж в інших смартфонів), вбудовану 2-мегапіксельну камеру і важив 136 г. Коли користувач підносив телефон до вуха, датчик наближення виявляв це і вимикав підсвічування екрана і сприйняття дотиків екрана на час розмови. У пристрої також був передбачений датчик зовнішньої освітленості для автоматичного налаштування яскравості екрана і акселерометр, який визначав вибір портретної або альбомної орієнтації екрана.

iPhone 1 містив 128 Мбайт оперативної пам’яті і 4, 8 або 16 Гбайт флеш-пам’яті та підтримував стандарт глобальної системи мобільного зв’язку ((Global System for Mobile communications, GSM), Wi-Fi (802.1 Ib/g) і Bluetooth.

На відміну від великої кількості загальнодоступної інформації про IBM PC, компанія Apple, як правило, не розкривала деталі архітектури iPhone. Apple не опублікувала жодної інформації про процесор або інші внутрішні компоненти першого iPhone, просто назвавши його закритою системою.

Попри відсутність офіційної інформації від Apple, знайшлися ентузіасти, які розібрали різні моделі iPhone і спробували ідентифікувати компоненти телефону та взаємозв’язки між ними. Дослідники програмного забезпечення розробили різні тести, які намагалися визначити конкретну модель процесора та інших цифрових пристроїв, встановлених в iPhone. Такі зусилля зі зворотного розроблення схильні до

помилок, тому описи архітектури iPhone у цьому розділі слід сприймати з певною часткою скептицизму.

В iPhone 1 був встановлений 32-розрядний процесор ARM11 з тактовою частотою 412 МГц, який випускає компанія Samsung. ARM11 був поліпшеним варіантом процесорів ARM попереднього покоління і містив 8-ступінчастий конвеєр оброблення інструкцій і підтримку моделі оброблення з **одним потоком інструкцій і безліччю потоків даних** (single instruction, multiple data, SIMD) для поліпшення якості звуку та відео. Архітектуру процесора ARM ми детально обговоримо в *розділі 10*.

Живлення iPhone 1 забезпечував літій-іонний полімерний акумулятор напругою 3,7 В. Заміна акумулятора не передбачалася, і, за оцінками Apple, після 400 циклів заряджання і розряджання він мав втратити близько 20% своєї первісної ємності. Apple заявила про те, що пристрій працюватиме до 250 годин у режимі очікування і до 8 годин у режимі розмови без підзарядки.

Через 6 місяців після появи iPhone журнал «Time» назвав iPhone «Винаходом року» за 2007 р. У 2017 р. «Time» склав рейтинг 50 *найвпливовіших пристроїв усіх часів*. iPhone очолив цей список.

У наступному розділі ми розглянемо взаємозв'язок розвитку технологічних досягнень у галузі обчислювальної техніки та фізичних обмежень інтегральних схем на базі кремнію, які лежать у їхній основі.

Закон Мура

Для тих, хто працює в галузі комп'ютерних технологій, що швидко розвивається, складання планів на майбутнє є складним завданням. Це справедливо незалежно від того, чи ставите ви перед собою мету спланувати власну кар'єру, чи розробити оптимальний графік інвестицій у дослідження і розробки для великої напівпровідникової корпорації. Ніхто і ніколи не може бути повністю впевненим, яким буде наступний технологічний стрибок, які з його наслідків вплинуть на галузь і користувачів або коли він відбудеться. Один із підходів, який виявився корисним у цих складних умовах, полягає в розробленні емпіричного правила або закону, заснованого на досвіді.

У 1957 р. Гордон Мур став співзасновником компанії Fairchild Semiconductor, а пізніше обіймав посади голови Ради директорів і генерального директора Intel. У 1965 р. Мур опублікував у журналі «Electronics» статтю, де він запропонував свій прогноз змін, які відбудуться в напівпровідниковій промисловості протягом наступних 10 років. У статті він зазначив, що кількість дискретних компонентів, як-от транзистори, діоди та конденсатори, які можна було розмістити на одному чіпі, подвоювалася приблизно щороку, і що ця тенденція, ймовірно, збережеться протягом наступних 10 років. Ця формула подвоєння стала відома як **закон Мура**. Це не був науковий закон на кшталт закону всесвітнього тяжіння. Радше це був результат спостереження за

історичними тенденціями, і Мур вирішив, що таке формулювання може певним чином прогнозувати майбутнє.

Для тих десяти років закон Мура виявився вражаюче точним. У 1975 р. він переглянув прогнозований темп зростання на наступне десятиліття, припустивши, що кількість компонентів на інтегральній схемі буде подвоюватися кожні два роки, а не щорічно. Цей темп витримувався десятиліттями – приблизно до 2010 р. В останні роки темпи зростання дещо знизилися. У 2015 р. Брайан Кржаніч, генеральний директор корпорації Intel, заявив, що темпи зростання компанії сповільнюються і подвоєння досягається приблизно кожні два з половиною роки.

Незважаючи на те що термін подвоєння щільності інтегральних схем збільшується, нинішні темпи зростання являють собою феноменальний прогрес, який, як можна очікувати, продовжиться і в майбутньому, але не так швидко, як раніше.

Закон Мура протягом десятиліть зарекомендував себе як надійний інструмент для оцінки діяльності напівпровідникових компаній.

Компанії використовували його для постановки цілей щодо продуктивності своїх продуктів і для планування своїх інвестицій. Порівнюючи збільшення щільності інтегральних схем для продуктів компанії з попередніми значеннями і з показниками інших компаній, керівники напівпровідникових компаній і галузеві аналітики можуть оцінювати і кваліфікувати ефективність компанії. Результати такого аналізу безпосередньо вплинули на рішення про інвестування у величезні нові заводи з виробництва інтегральних схем і досягнення нових рівнів інтеграції за рахунок зменшення розмірів елементів.

За десятиліття, що минули з моменту появи IBM PC, можливості однокристальних мікропроцесорів значно розширилися. Процесори сучасних поколінь у сотні разів швидші, від самого початку працюють із 32- і 64-бітними даними, мають набагато більше вбудованої пам’яті та розкривають набагато більше функціональних можливостей, і все це – на одній інтегральній схемі.

Ці удосконалення стали можливими завдяки збільшенню щільності розміщення напівпровідникових елементів, як передбачає закон Мура. Транзистори меншого розміру працюють на вищих тактових частотах через коротші з’єднання між елементами схеми. Крім того, менші транзистори, очевидно, дають змогу реалізувати більше функцій на заданій площі кристала. Через менші розміри та відстані між сусідніми компонентами транзистори споживають менше енергії та виділяють менше тепла.

У законі Мура не було нічого магічного. Це було лише спостереження за тенденціями того часу. Однією з тенденцій було неухильне зростання розмірів напівпровідникових кристалів. Це стало результатом поліпшення виробничих процесів, які знизили щільність дефектів, що дало змогу отримати прийнятний вихід продукції з більшими

кристалами інтегральних схем. Іншою тенденцією було постійне зменшення розміру найменших компонентів, які можна було надійно виготовити в складі схеми. Остання тенденція була зумовлена тим, що Мур називав «винахідливістю» розробників схем, що виражалася у дедалі ефективнішому та дієвішому використанні дедалі більшої кількості схемних елементів, розміщених на кристалі.

Традиційні процеси виробництва напівпровідників почали наближатися до фізичних меж, які в кінцевому рахунку загальмують зростання, передбачене законом Мура. Найменші елементи сучасних комерційно доступних інтегральних схем мають розмір близько 5 **нанометрів (нм)**. Для порівняння: типова людська волосина має товщину близько 50 000 нм, а молекула води (одна з найменших молекул) має розмір 0,28 нм. Існує межа, що обмежує подальше зменшення схемних елементів, оскільки їхні розміри наближаються до атомарного масштабу.

На додаток до проблеми створення надійних елементів з невеликої кількості молекул починають проявлятися й інші фізичні ефекти, як-от дифракційна *межа Аббе*, які можуть стати серйозною перешкодою для виробництва одноразрядних схем нанометрового масштабу.

Ми не будемо вдаватися до подробиць цих явищ, достатньо знати, що неухильне збільшення щільності компонентів інтегральних схем, яке тривало протягом десятиліть відповідно до закону Мура, найближчими роками стане набагато важче підтримувати.

Це не означає, що ми «застрягнемо» на рівні процесорів, які будуть багато в чому аналогічні тим, що зараз доступні на ринку. Попри уповільнення зростання щільності транзисторів, виробники напівпровідників використовують низку альтернативних методів для подальшого збільшення потужності обчислювальних пристроїв. Одним із підходів є спеціалізація, за якої схеми розробляються для високоефективного розв’язання задач певної категорії, а не просто для задовільного виконання широкого кола завдань.

Графічні процесори (graphics processing units, GPU) є чудовим прикладом спеціалізації. Графічні процесори першого покоління були націлені виключно на підвищення швидкості побудови тривимірних графічних сцен, в основному для використання у відеоіграх. Розрахунки, необхідні для побудови тривимірної сцени, добре визначені і повинні застосовуватися до тисяч пікселів для створення одного кадру. Цей процес повторюється для кожного наступного кадру, а щоб забезпечити задовільну взаємодію з користувачем, кадри повинні перемальовуватися з частотою 60 Гц або вище. Вимогливий до обчислювальних ресурсів і циклічний характер цього завдання ідеально підходить для прискорення за допомогою апаратного розпаралелювання. Кілька обчислювальних модулів усередині графічного процесора одночасно виконують практично одні й ті самі обчислення з різними вхідними даними для отримання окремих потоків вихідних даних. Ці вихідні потоки об’єднуються для

створення повної сцени. Архітектура сучасних графічних процесорів була вдосконалена для підтримки інших сфер застосування обчислень, таких як навчання нейронних мереж на великих обсягах даних. Архітектури графічних процесорів будуть детально описані в *розділі 6*.

Найближчими роками тенденція, описана законом Мура, продемонструє ознаки ослаблення. То які досягнення можуть компенсувати цей спад, щоб запустити наступний раунд інновацій у комп’ютерних архітектурах? Сьогодні ми не знаємо напевно, але деякі привабливі варіанти нині пильно вивчаються. Квантові обчислення це один із прикладів таких технологій. Ми розглянемо цю технологію *в розділі 17*.

Квантові обчислення використовують властивості субатомних частинок для опрацювання даних у такий спосіб, який недоступний для традиційних комп’ютерів. Основним елементом квантових обчислень є кубіт, або квантовий біт. Кубіт подібний до звичайного двійкового біта, але крім представлення станів 0 і 1, кубіти можуть досягати стану, що є суперпозицією (або сумішшю) станів 0 і 1. Під час вимірювання вихідне значення кубіта завжди дорівнюватиме 0 або 1, але ймовірність отримання будь-якого з вихідних значень залежить від квантового стану кубіта перед зчитуванням. Для того щоб скористатися унікальними можливостями квантових обчислень, необхідні спеціалізовані алгоритми.

Інша перспективна можливість полягає в тому, що наступний значний технологічний прорив в обчислювальних пристроях буде пов’язаний із чимось, про що ми або не думали, або, якщо ми думали про це, то могли відразу відкинути цю ідею як нереалістичну. iPhone, який обговорювали в попередньому розділі, є прикладом інноваційного продукту, що створив нову категорію, – цей пристрій зробив революцію в особистому спілкуванні та відкрив нові способи використання Інтернету. Наступним великим досягненням може бути новий тип продукту, дивовижна нова технологія або якесь поєднання продукту і технології. Прямо зараз ми не знаємо, що це буде і коли це станеться, але ми з упевненістю можемо сказати, що такі зміни настають.

У наступному розділі представлено деякі фундаментальні концепції цифрових обчислень, які необхідно зрозуміти, перш ніж ми заглибимося в цифрові схеми і деталі архітектури сучасних комп’ютерів у наступних розділах.

Архітектура комп’ютерів

Описи низки ключових архітектур з історії обчислювальної техніки, представлені в попередніх розділах цього розділу, містили деякі знайомі або незнайомі вам терміни. У цьому розділі буде представлено концептуальні структурні елементи, які використовуються для створення сучасних процесорів і пов’язаних з ними комп’ютерних підсистем.

Представлення чисел рівнями напруги

Однією із загальноприйнятих особливостей сучасних комп'ютерів є використання рівнів напруги для зазначення значень даних. Зазвичай розпізнаються тільки два рівні напруги: низький і високий. Низькому рівню часто присвоюють значення 0, а високому рівню – значення 1. Напруга в будь-якій точці ланцюга (цифровому чи іншому) має аналогову природу і може приймати будь-яке значення в межах свого робочого діапазону.

Під час переходу з низького рівня на високий або назад напруга має проходити через усі проміжні рівні. У контексті цифрових схем переходи між низькими та високими рівнями відбуваються швидко, і ці схеми проєктують таким чином, щоб не реагувати на напруги між високим і низьким рівнями.

Двійкові та шістнадцяткові числа

Схеми всередині процесора в жодному з можливих сенсів не працюють безпосередньо з числами. Схемні елементи процесора підкоряються законам електрики та електроніки і просто реагують на вхідні сигнали, що надходять. Вхідні сигнали, які керують цими діями, породжуються кодом, розробленим програмістами, і даними, наданими як вхідні дані для програми. Інтерпретація результатів роботи програми, наприклад чисел в електронній таблиці або символів у програмі опрацювання текстів, це суто людська інтерпретація, що надає значення результату електронних взаємодій усередині процесора. Рішення присвоїти 0 низькій напрузі та 1 високій напрузі є першим кроком у процесі цієї інтерпретації.

Найменшою одиницею інформації в цифровому комп'ютері є двійкова цифра, звана бітом, яка являє собою окремий елемент даних, що містить значення 0 або 1. Кілька бітів можна згрупувати, щоб забезпечити представлення більшого діапазону значень. Байт складається з 8 бітів, об'єднаних в одне значення. Байт – це найменша одиниця інформації, яка може бути прочитана або записана в пам'ять більшістю сучасних процесорів. Деякі комп'ютери минулого і сьогодення використовують різну кількість бітів для найменшого адресованого елемента даних, але 8-бітний байт представляє найпоширеніший розмір.

Один біт може набувати двох значень: 0 або 1. Два згруповані біти можуть набувати чотирьох значень: 00, 01, 10 і 11. Три біти можуть приймати вісім значень: 000, 001, 010, 011, 100, 101, 110 і 111. Загалом група з n бітів може набувати 2^n значень. Таким чином, 8-бітний байт може представляти 2^8 або 256 унікальних значень.

Коли справа доходить до виконання арифметичних операцій, двійковий формат подання чисел не є кращим для більшості людей. Робота з числами, представленими у вигляді 11101010, може бути доволі складною і схильною до помилок, особливо якщо йдеться про маніпулювання 32- і 64-бітними

значеннями. Для того щоб спростити роботу з цими числами, часто замість них використовують **шістнадцяткові числа**. Термін "шістнадцятковий" (hexadecimal) часто скорочують до *hex*.

У шістнадцятковій системі числення двійкові числа розділені на групи по чотири біти. При 4 бітах у групі кількість можливих значень дорівнює 2^4 або 16. Перші десять із цих 16 чисел позначаються цифрами від 0 до 9, а решта шість – буквами від A до F. У табл. 1.1 показано перші 16 двійкових значень, починаючи з 0, разом із відповідною шістнадцятковою цифрою та десятковим еквівалентом двійкового та шістнадцяткового значень.

Таблиця 1.1. *Двійкові, шістнадцяткові та десяткові числа*

Двійкова	Шістнадцяткова	Десяткова
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Двійкове число 11101010 можна подати компактніше, розбивши його на дві групи по чотири біти (1110 та 1010) і записавши їх у вигляді шістнадцяткових цифр EA. Групу з чотирьох бітів іноді називають *напівбайтом*, тобто половиною байта. Двійкові числа можуть набувати тільки двох значень, тому двійкова система є системою числення з основою 2. Шістнадцяткові числа можуть приймати 16 значень, тому шістнадцяткова система числення має основу 16. Десяткові числа можуть мати 10 значень, тому десяткова система має основу 10.

Під час роботи з різними системами числення легко заплутатися. Чи є число, записане як 100, двійковим, шістнадцятковим або десятковим? Без додаткової інформації відповісти на це запитання неможливо. Різні мови програмування і підручники використовували різні підходи до усунення подібної двозначності. У більшості випадків десяткові числа не мають додаткових позначень, тому число, записане як 100, зазвичай є десятковим. У мовах програмування, таких як C і C++, перед шістнадцятковими числами ставиться позначення *0x*, тому число у вигляді *0x100* являє собою 100 у шістнадцятковому форматі. У мовах асемблера для позначення шістнадцяткових чисел можуть використовуватися або префікс *$*, або суфікс h. Використання двійкових значень у програмуванні менш поширене – здебільшого це зумовлено тим, що шістнадцяткові значення є кращими через їхню компактність. Деякі компілятори підтримують використання *0b* як префікса для двійкових чисел.

ПРЕДСТАВЛЕННЯ ШЕСТНАДЦАТЕРИЧНОГО ЧИСЛА



У цій книжці для представлення шістнадцяткових чисел, залежно від контексту, використовується або префікс *$*, або суфікс *h*. Суфікс *b* позначає двійкові числа, а відсутність префікса або суфікса вказує на десяткові числа.

Біти в двійковому числі нумеруються індивідуально, причому біт 0 у крайній правій позиції є найменш значущим бітом. Номери бітів збільшуються за величиною справа наліво, аж до найстаршого біта в крайній лівій позиції.

Деякі приклади допоможуть прояснити ці правила. У табл. 1.1 у двійковому значенні 0001b (1 у десятковому форматі) для біта з номером 0 встановлено 1, а інші три біти дорівнюють 0. Для 0010b (2 у десятковому форматі) біт 1 встановлено, а інші біти скинуто. Для 0100b (4 у десятковому форматі) біт 2 встановлено, а інші біти скинуто.

ВСТАНОВЛЕНИЙ БІТ І СКИНУТИЙ БІТ



Встановлений біт має значення 1. Скинутий біт має значення 0.

8-бітний байт може приймати значення від $00h до $FF, еквівалент десяткового діапазону 0-255. Під час виконання додавання на рівні байтів результат може перевищувати 8 бітів. Наприклад, додавання $01 до $FF дає результат $100. У разі використання 8-бітних регістрів це означає операцію перенесення в дев’ятий біт, яку потрібно відповідним чином обробляти апаратними засобами процесора і програмним забезпеченням, що виконує додавання.

У беззнаковій арифметиці віднімання $01 з $00 дає значення $FF, тобто виконується циклічний перехід до значення $FF. Залежно від виконуваних обчислень це може бути бажаним результатом, а може і не бути ним. Знову ж таки, апаратні засоби і програмне забезпечення процесора повинні обробити цю ситуацію, щоб прийти до бажаного результату.

За необхідності від'ємні значення можуть бути представлені за допомогою двійкових чисел. Найпоширенішим форматом чисел зі знаком у сучасних процесорах є **додатковий код**. При його використанні 8-бітові числа зі знаком охоплюють діапазон від -128 до 127. Старший біт значення в додатковому коді є бітом знака: 0 у цьому біті представляє додатне число, а 1 – від'ємне число. Заперечення числа в додатковому коді (множення на -1) виконується шляхом інвертування всіх його бітів, додавання 1 та ігнорування перенесення. Інверсія біта означає зміну стану скинутого біта на 1, а встановленого біта – на 0. Низку покрокових прикладів заперечення 8-бітних чисел зі знаком наведено в табл. 1.2.

Таблиця 1.2. *Приклади операції заперечення*

Десяткове значення	Двійкове значення	Інверсія бітів	Додавання одиниці	Результат заперечення
0	00000000b	11111111b	00000000b	0
1	00000001b	11111110b	11111111b	-1
-1	11111111b	00000000b	00000001b	1
127	01111111b	10000000b	10000001b	-127
-127	10000001b	01111110b	01111111b	127

Заперечення 0 повертає 0, як і слід було очікувати з точки зору математики.

АРИФМЕТИКА ДОДАТКОВОГО КОДУ



На рівні бітів арифметика додаткового коду ідентична беззнаковій арифметиці. Маніпуляції, пов'язані зі складанням і відніманням, однакові незалежно від того, чи є вхідні значення знаковими або беззнаковими. Інтерпретація результату як знакового або беззнакового повністю залежить від наміру користувача.

У табл. 1.3 показано відповідність двійкових значень від 00000000b до 11111111b значенням зі знаком у діапазоні від -128 до 127 і беззнаковим значенням від 0 до 255.

Таблиця 1.3. *Знакові та беззнакові 8-бітові числа*

Двійкова	Десяткова з знаком	Десяткова без знака
00000000b	0	0
00000001b	1	1
00000010b	2	2
…	…	…
01111110b	126	126
01111111b	127	127
10000000b	-128	128
10000001b	-127	129
…	…	…
11111101b	-3	253
11111110b	-2	254
11111111b	-1	255

Знакові та беззнакові подання двійкових чисел поширюються на більші цілочисельні типи даних. 16-бітові значення можуть представляти цілі числа без знака від 0 до 65 535 і цілі числа зі знаком у діапазоні від -32 768 до 32 767. У сучасних процесорах і мовах програмування зазвичай доступні 32-бітні, 64-бітні і навіть більші цілочисельні типи даних.

Микропроцессор 6502

У цьому розділі представлено процесорну архітектуру, яка є відносно простою порівняно з більш потужними сучасними процесорами.

Мета такого підходу – представлення деяких базових концепцій, спільних для широкого кола процесорів, від бюджетних мікроконтролерів до складних багатоядерних 64-розрядних процесорів.

Процесор 6502 був представлений компанією MOS Technology в 1975 р. Він знайшов широке застосування в ігрових консолях від Atari і Nintendo, а також у комп'ютерах, які продаються Commodore і Apple. Різні версії процесора 6502, як і раніше, широко використовуються у вбудованих системах і сьогодні – згідно з деякими оцінками, з 2018 р. їх випущено від 5 до 10 млрд (так-так, мільярдів). У поп-культурі, і робот Бендер з «Футурами», і Т-800 з «Термінатора», судячи з низки кадрів, працювали на 6502.

Як і багато ранніх мікропроцесорів, 6502 використовував джерело живлення напругою 5 вольт (В) постійного струму. У таких схемах низький рівень сигналу – це будь-яка напруга від 0 до 0,8 В, а високий рівень – будь-яка напруга від 2 до 5 В. Напруги між цими діапазонами виникають тільки під час переходів від низького до високого і від високого до низького рівня. Низький рівень сигналу визначається як логічний 0, а високий рівень сигналу – як логічна 1.

У *розділі* 2 будуть детально розглянуті електронні схеми, що використовуються в цифровій електроніці.

Довжина слова процесора визначає розмір основного елемента даних, з яким працює процесор. Довжина слова процесора 6502 дорівнює 8 бітам. Це означає, що 6502 зчитує і записує дані в пам’ять по 8 біт за раз і зберігає дані у 8-розрядних регістрах.

Для пам’яті програм і пам’яті даних використовується один і той самий адресний простір, а доступ до пам’яті 6502 здійснюється по одній шині. Як і в Intel 8088, у 6502 реалізовано архітектуру фон Неймана. Цей процесор має 16-розрядну адресну шину, що дає змогу адресувати 64 Кбайт пам’яті.

1 Кбайт містить 2^{10} або 1024 байти. Кількість унікальних двійкових комбінацій для 16 адресних рядків дорівнює 2^{16}, що дає змогу отримати доступ до 65 536 комірок пам’яті з байтовою організацією. Зверніть увагу, що здатність пристрою адресувати 64 Кбайт не означає, що кожній із цих комірок має відповідати фізична пам’ять. Комп’ютер Commodore VIC-20, заснований на процесорі 6502, містив усього 5 Кбайт ОЗП і 20 Кбайт ПЗП.

Процесор 6502 містить внутрішні області зберігання даних, звані регістрами. **Регістр** – це місце в логічному пристрої, в якому слово інформації може зберігатися і оброблятися під час обчислень. Типовий процесор містить невелику кількість регістрів для тимчасового зберігання значень даних і виконання таких операцій, як додавання або обчислення адреси.

На рис. 1.1 показано структуру регістрів 6502. Процесор містить п’ять 8-розрядних регістрів (A, X, Y, P і S) і один 16-розрядний регістр (PC). Цифри над кожним регістром вказують номери бітів на кожному з кінців регістра.

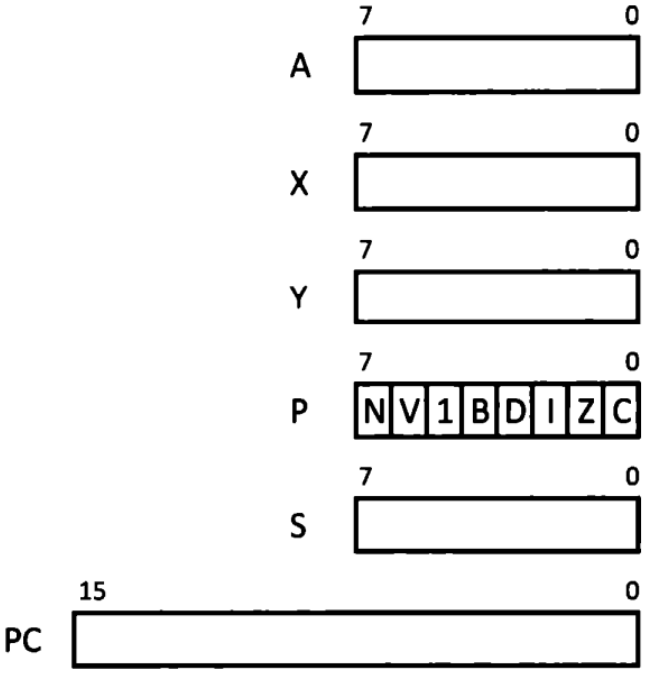


Рис 1.1. *Набір регістрів процесора 6502*

Кожен із регістрів A, X і Y може слугувати сховищем загального призначення. Інструкції програми можуть завантажувати значення в один із цих регістрів і після виконання кількох інших інструкцій використовувати збережене значення для будь-якої мети, якщо проміжні інструкції не змінили вміст цього регістра. Регістр A – це єдиний регістр, що підтримує виконання арифметичних операцій. Регістри X і Y, але не регістр A, можна використовувати як індексні регістри під час обчислення адрес пам'яті.

Регістр P містить прапори процесора. Кожен біт у цьому регістрі має унікальне призначення, за винятком біта, позначеного 1. Біт 1 не використовується і може бути проігнорований. Інші біти в цьому регістрі називаються **прапорами** і вказують на певний стан або являють собою параметр конфігурації. У процесорі 6502 використовуються такі прапори.

- N – прапор від'ємного знака. Цей прапор встановлюється, коли в результаті арифметичної операції встановлюється біт 7. Використовується у знаковій арифметиці.
- V – прапор переповнення. Він встановлюється, коли операція додавання або віднімання над числами зі знаком призводить до позитивного або негативного переповнення – виходу за межі діапазону від -128 до 127.
- B – прапор програмного переривання. Він вказує на виконання інструкції Break (brk). Цей біт відсутній у самому регістрі P. Значення прапора B враховується тільки при перевірці вмісту регістра P, збереженого в стеку за допомогою інструкції BRK або переривання. Прапор B встановлюється для того, щоб під час обробки переривань відрізняти програмне переривання, що виникло в результаті виконання інструкції BRK, від апаратного переривання.
- D – прапор десяткового режиму. Встановлення цього прапора означає, що арифметика процесора працюватиме в двійково-десятковому (binary-coded decimal, BCD) режимі. Режим BCD використовується рідко і тут обговорюватися не буде. Слід зазначити, що цей режим обчислень з основою 10 нагадує архітектури аналітичної машини Беббіджа та ENIAC.
- I – прапор заборони переривань. Його встановлення вказує, що входи переривань (крім немаскованого переривання) оброблятися не будуть.
- Z – прапор нуля. Він встановлюється, коли операція призводить до результату, що дорівнює 0.
- C – прапор перенесення. Він встановлюється, коли арифметична операція призводить до перенесення.

Прапори N, V, Z і C є найбільш важливими прапорами в контексті загальних обчислень, що включають цикли, підрахунок і арифметику.

Регістр S є **покажчиком стека**. У процесорі 6502 стек – це область пам’яті, розташована за адресами від $100 до $1FF. Ця область довжиною 256 байтів використовується для тимчасового зберігання параметрів підпрограм і містить зворотну адресу при виклику підпрограми. Під час запуску системи регістр S ініціалізується таким чином, щоб вказувати на верхню частину цієї області. Значення «проштовхуються» в стек за допомогою таких інструкцій, як PHA, яка поміщає в стек вміст регістра A.

Коли значення поміщається в стек, процесор 6502 зберігає це значення за адресою, вказаною регістром S, після додавання фіксованого зміщення $100, а потім зменшує значення регістра S на одиницю. Додаткові значення можуть бути поміщені в стек шляхом виконання додаткових інструкцій запису в стек. У міру введення додаткових значень стек у пам’яті зростає в напрямку вниз. Програми повинні стежити за тим, щоб не перевищувати зафіксований для стека розмір у 256 байт під час завантаження в нього даних.

Дані, що зберігаються в стеку, потрібно витягувати в порядку, зворотному до того, в якому вони були поміщені в стек. Стек являє собою структуру даних, організовану за принципом **«останнім увійшов, першим вийшов»** (last in, first out, LIFO). Це означає, що значення, яке витягується зі стека, – це байт, який був доданий у нього останнім. Інструкція PLA збільшує значення регістра S на 1, потім копіює значення за адресою, вказаною регістром S (плюс зміщення $100), у регістр A.

Регістр PC – це лічильник інструкцій. Він містить адресу інструкції, яка має бути виконана наступною. На відміну від інших регістрів, PC має розмір 16 біт, що забезпечує доступ до всього адресного простору процесора 6502.

Кожна інструкція складається з 1-байтового коду операції, для стислості званого **опкодом** (opcode), і може супроводжуватися байтами операндів (від 0 до 2), залежно від типу інструкції. Після виконання кожної інструкції вміст регістра PC оновлюється, щоб вказати на інструкцію, наступну за щойно виконаною. На додаток до автоматичних оновлень під час послідовного виконання інструкцій регістр PC може бути змінений інструкціями переходу, розгалуження, виклику підпрограм і повернення з них.

Набір інструкцій мікропроцесора 6502

Тепер розглянемо набір інструкцій процесора 6502. Інструкції – це окремі команди процесора, які вибудовуються в послідовність для виконання алгоритму, закодованого програмістом. Інструкція містить двійкове число, зване кодом операції (або опкодом), яке повідомляє процесору, що слід робити під час виконання цієї інструкції.

За бажання програмісти можуть писати код безпосередньо за допомогою інструкцій процесора. Приклади цього ми побачимо пізніше в цьому розділі. Програмісти також можуть створювати код так званою мовою високого рівня. Потім програміст

використовує програмний інструмент, званий компілятором, який перетворює високорівневий код на послідовність інструкцій процесора (зазвичай набагато довшу).

У цьому розділі ми працюємо з кодом, написаним у вигляді послідовностей інструкцій процесора. Ця форма вихідного коду називається **мовою асемблера**.

Кожна інструкція процесора 6502 має трисимвольне мнемонічне позначення. У вихідних файлах мови асемблера кожен рядок коду містить мнемонічне позначення інструкції, за яким слідують будь-які операнди, які стосуються цієї інструкції. Комбінація мнемонічного позначення й операндів визначає **режим адресації**. Процесор 6502 підтримує кілька режимів адресації, забезпечуючи гнучкість доступу до даних у регістрах і пам'яті. Для цього введення ми працюватимемо тільки з режимом **безпосередньої** адресації, у якому сам операнд містить значення, а не вказує регістр або комірку пам'яті, що містить потрібне значення. Безпосередньому значенню передує символ #.

В асемблері процесора 6502 десяткові числа не мають додаткових позначень (48 означає число 48 у десятковому форматі), тоді як шістнадцятиричним значенням передує символ $ ($30 означає число 30 у шістнадцятковому форматі, еквівалент 00110000b і десяткового числа 48). Безпосереднє десяткове значення виглядає як #48, а шістнадцяткове значення – як #$30.

Деякі приклади асемблерного коду продемонструють арифметичні можливості процесора 6502. У таких прикладах використовуються п'ять інструкцій 6502:

- LDA завантажує значення в регістр A;
- ADC виконує додавання з урахуванням перенесення (прапор C у регістрі P) як додаткові вхідні та вихідні дані;
- SBC виконує віднімання з урахуванням прапора C як додаткових вхідних і вихідних даних;
- SEC встановлює прапор C безпосередньо;
- CLC скидає прапор C безпосередньо.

Оскільки прапор C надає вхідні дані для інструкцій додавання і віднімання, перед виконанням інструкції ADC або SBC важливо переконатися, що він має правильне значення. Перед виконанням операції додавання прапор C має бути скинутий, щоб вказати на відсутність перенесення з попередньої операції додавання. Під час додавання багатобайтових значень (наприклад, під час роботи з 16-, 32- або 64-бітними числами) перенесення, якщо воно є, поширюватиметься від суми однієї пари байтів до наступної в міру додавання значущих байтів. Якщо прапор C встановлено під час виконання інструкції ADC, то до результату слід додати 1. Після виконання інструкції ADC прапор C виступає як дев'ятий біт результату: якщо прапор C має значення 0, перенесення не було, а 1 вказує на те, що з 8-розрядного регістра було виконано перенесення.

Віднімання з використанням інструкції SBC може трохи збити з пантелику початківців-програмістів мовою асемблера процесора 6502. Школярі, які вивчають віднімання, використовують техніку запозичення під час віднімання більшої цифри з меншої. У процесорі 6502 прапор C являє собою ознаку, протилежну запозиченню. Якщо C має значення 1, то позика дорівнює 0, а якщо C дорівнює 0, то позика дорівнює 1. Виконання простого віднімання без вхідного запозичення вимагає встановлення прапора C перед виконанням інструкції SBC.

У прикладах із табл. 1.4 процесор 6502 використовується як калькулятор із вхідними даними, визначеними у вигляді безпосередніх значень у коді, і з результатом, що зберігається в регістрі A. Стовпчик "Результати" містить остаточне значення регістра A і стани прапорів N, V, Z і C.

Таблиця 1.4. *Послідовності арифметичних інструкцій процесора 6502*

Послідовність інструкцій	**Опис**	**Результати**				
		A	**N**	**V**	**Z**	**C**
CLC LDA #1 ADC #1	8-бітове додавання без перенесення: скидання прапора перенесення, потім завантаження безпосереднього значення 1 у регістр A і додавання до нього 1	$02	0	0	0	0
SEC LDA #1 ADC #1	8-бітове додавання з перенесенням: встановлення прапора перенесення, потім завантаження безпосереднього значення 1 у регістр A і додавання до нього 1	$03	0	0	0	0
SEC LDA #1 SBC #1	8-бітове віднімання без запозичення: встановлення прапора перенесення, потім завантаження безпосереднього значення 1 у регістр A і віднімання з нього 1. C = 1 означає відсутність запозичення	$00	0	0	1	1
CLC LDA #1 SBC #1	8-бітове віднімання із запозиченням: скидання прапора перенесення, потім завантаження безпосереднього значення 1 у регістр A і віднімання з нього 1. C = 0 вказує на наявність запозичення	$FF	1	0	0	0
CLC LDA $FF ADC #1	Позитивне переповнення без знака: додавання 1 до $PP. C = 1 вказує на наявність переносу	$00	0	0	1	1
SEC LDA #0 SBC #1	Негативне переповнення без знака: віднімання 1 з 0. C = 0 вказує на наявність запозичення	$FF	1	0	0	0
CLC LDA #$7F ADC #1	Позитивне переповнення зі знаком: додавання 1 до $7P. C = 1 вказує на наявність позитивного переповнення зі знаком	$80	1	1	0	0

Таблиця 1.4. *(закінчення)*

Послідовність інструкцій	Опис	Результати				
		A	N	V	Z	C
SEC LDA #$80 SBC #1	Негативне переповнення зі знаком: віднімання 1 від $80. C = 1 вказує на наявність від'ємного переповнення зі знаком	$7F	0	1	0	1

Якщо у вас під рукою немає комп'ютера на базі процесора 6502 з асемблером і налагоджувачем, в Інтернеті є кілька його безкоштовних емуляторів, які ви можете запустити у своєму веб-браузері. Один чудовий емулятор доступний за адресою **https://skilldrick.github.io/easy6502/**. Зайдіть на цей веб-сайт і прокрутіть вміст вікна донизу, доки не знайдете лістинг коду за замовчуванням із кнопками для складання та запуску коду 6502. Замініть лістинг коду за замовчуванням групою з трьох інструкцій із табл. 1.4, потім виконайте код. Для того щоб вивчити дію кожної інструкції в послідовності, використовуйте елементи керування відладчика для покрокового виконання інструкцій і спостереження за результатом виконання кожної інструкції в регістрах процесора.

Для того щоб вивчити дію кожної інструкції в послідовності, використовуйте елементи керування відладчика для покрокового виконання інструкцій і спостереження за результатом виконання кожної інструкції в регістрах процесора.

У цьому розділі представлено дуже коротке введення в процесор 6502 і розглянуто лише невелику частину його можливостей. Одна з цілей цього розгляду полягала в тому, щоб проілюструвати складність розв'язання проблеми перенесення при виконанні додавання і запозичень при виконанні віднімання. Починаючи з Чарльза Беббіджа і закінчуючи проектувальниками процесора 6502 і розробниками сучасних обчислювальних систем, архітектори комп'ютерів знаходили розв'язання задач обчислень і втілювали їх у життя із застосуванням найкращих доступних їм технологій.

Резюме

Цей розділ розпочався з короткої історії автоматичних обчислювальних пристроїв, за якою послідував огляд значних технологічних досягнень, що призвели до низки проривів в обчислювальних можливостях. Потім було обговорення закону Мура з оцінкою його застосовності в попередні десятиліття і можливих наслідків для майбутнього. Основні поняття комп'ютерної архітектури були представлені на прикладі обговорення регістрів і набору інструкцій мікропроцесора 6502. Історія комп'ютерної архітектури захоплива, і я рекомендую вам вивчити її детальніше.

У наступному розділі ми познайомимося з цифровою логікою, починаючи з властивостей основних електричних схем і закінчуючи проектуванням цифрових

підсистем, що використовуються в сучасних процесорах. Ви дізнаєтеся про будову логічних вентилів, тригерів і цифрових схем, включно з мультиплексорами, регістрами зсуву і суматорами. Цей розділ включає введення в мови опису апаратних засобів, які є спеціалізованими комп'ютерними мовами, що застосовуються при проектуванні складних цифрових пристроїв, таких як комп'ютерні процесори.

Вправи

1. Використовуючи свою улюблену мову програмування, розробіть модель однорозрядного десяткового суматора, який працює так само, як суматор аналітичної машини Беббіджа. Спочатку запитайте в користувача дві цифри в діапазоні 0-9: доданок і акумулятор. Відображення доданка, акумулятора та ознаки перенесення, яка спочатку дорівнює 0. Виконайте серію циклів у такий спосіб:
   1) якщо доданок дорівнює 0, виведіть на екран значення доданка, акумулятора й ознаки перенесення, після чого завершіть програму;
   2) зменшіть доданок на 1 і збільште значення акумулятора на 1 ;
   3) якщо значення акумулятора збільшується з 9 до 0, встановіть ознаку перенесення;
   4) поверніться до кроку 1.

   Протестуйте свій код на прикладі таких операцій додавання: 0 + 0, 0 + 1, 1 + 0, 1 + 2, 5 + 5, 9 + 1 та 9 + 9.
2. Створіть для доданка, акумулятора та ознак перенесення масиви з 40 десяткових цифр кожен. Запитайте в користувача два цілих десяткових числа довжиною до 40 цифр кожне. Виконайте додавання чисел, цифра за цифрою, використовуючи цикли, описані у вправі 1, і зберіть вихідні дані у вигляді ознак перенесення з позиції кожної цифри в масиві ознак перенесення. Після завершення циклів вставте перенесення і, за необхідності, розповсюдьте їх по цифрах, щоб завершити операцію додавання. Виводьте результати на екран після кожного циклу та після завершення. Проведіть тест із тими самими сумами, що й у *вправі 1*, а також перевірте суми 99 + 1, 999999 + 1, 49 + 50 та 50 + 50.
3. Змініть програми вправ 1 і 2 так, щоб реалізувати віднімання 40-значних десяткових значень. Виконуйте запозичення в міру необхідності. Перевірте програму на прикладі різниць 0 – 0, 1 – 0, 1000000 – 1 та 0 – 1. Який результат операції 0 – 1?
4. Мова асемблера процесора 6502 посилається на дані в комірках пам'яті, використовуючи значення операнда, що містить адресу (без символу #, який вказує на безпосереднє значення). Наприклад, інструкція LDA $00 завантажує в регістр A байт, що знаходиться в пам'яті за адресою $00. STA $01 зберігає байт

із регістра A за адресою $01. Адреси можуть мати будь-яке значення в діапазоні від 0 до $FFFFFF за умови, що за вказаною адресою є фізична пам'ять і ця адреса не використовується для будь-яких інших цілей. За допомогою кращого емулятора процесора 6502 напишіть код на асемблері 6502 для збереження 16-бітного значення за адресами $00-$01, збережіть друге значення за адресами $02-$03, потім додайте ці два значення і збережіть результат за адресами $04-$05. Забезпечте поширення переносів між двома байтами. Ігноруйте будь-яке перенесення з 16-бітного результату. Перевірте код на прикладах $0000 + $0001, $00FF + $0001 і $1234 + $5678.

5. Напишіть на асемблері 6502 код для віднімання двох 16-розрядних значень способом, аналогічним показаному у вправі 4. Перевірте код на прикладах $0001 – $0000, $0001 – $0001, $0100 – $00FF і $0000 – $0001. Який результат операції $0000 – $0001?

6. Напишіть на асемблері 6502 код для збереження двох 32-розрядних цілих чисел за адресами $00-$03 і $04-$07, а потім складіть їх, зберігши результати за адресами $08-$0B. Використовуйте циклічну конструкцію, що включає мітку і команду розгалуження, для перебору байтів двох значень, що складаються. Знайдіть в Інтернеті докладну інформацію про інструкції зменшення на 1 і розгалуження для процесора 6502, а також про використання міток у мові асемблера. Підказка: у цьому додатку добре працює реалізований у процесорі 6502 режим адресації з індексуванням нульової сторінки.

2

Цифрова логіка

Цей розділ ґрунтується на вступних темах, представлених у розділі 1, і дає чітке уявлення про цифрові структурні елементи, які використовуються під час проектування сучасних процесорів та інших складних електронних схем. Ми почнемо з обговорення базових елементів електричних схем. Далі познайомимося з транзисторами та дослідимо їхнє застосування як перемикальних пристроїв у простих логічних вентилях. Потім ми побудуємо з логічних вентилів засувки, тригери та кільцеві лічильники. Після цього шляхом об'єднання представлених раніше пристроїв створимо більш складні компоненти процесора, такі як регістри та суматори. Також тут буде введено поняття послідовної логіки, тобто логіки, побудованої на інформації про стан, яка змінюється з часом. Розділ закінчується введенням у мови опису апаратних засобів, що представляють кращий метод проектування складних цифрових пристроїв.

У цьому розділі будуть розглянуті такі теми:

- електричні схеми;
- транзистор;
- логічні вентилі;
- засувки;
- тригери;
- регістри;
- суматори;
- синхронізація;
- послідовна логіка;
- мови опису апаратних засобів.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-andOrganization-Second-Edition.**

Електричні схеми

Ми почнемо цей розділ з короткого огляду властивостей електричних ланцюгів.

Провідні матеріали, такі як мідь, мають здатність легко створювати електричний струм у присутності електричного поля. Непровідні матеріали, наприклад скло, гума і полівінілхлорид (ПВХ), перешкоджають проходженню електричного струму настільки сильно, що їх використовують як ізолятори для захисту електричних провідників від коротких замикань. У металах електричний струм складається з рухомих електронів. У конструкції резисторів використовуються матеріали, які допускають протікання деякого електричного струму струму, але при цьому передбачувано обмежують його допустиму величину.

Зв'язок між силою електричного струму, напругою і опором у ланцюзі аналогічний зв'язку між швидкістю потоку, тиском і обмеженням потоку в гідравлічній системі. Розглянемо кухонний водопровідний кран: тиск у трубі, що веде до крана, змушує воду текти, коли кран відкритий. Якщо кран відкрити зовсім трохи, вода буде текти з нього тонкою цівкою. Якщо кран відкрити більше, швидкість потоку збільшиться. Відкривання крана еквівалентно зменшенню опору потоку води, що протікає через кран.

В електричному ланцюзі напруга відповідає тиску у водопроводі. Електричний струм, сила якого вимірюється в амперах (часто скорочується до «А»), відповідає швидкості потоку води через трубу і кран. Електричний опір відповідає обмеженню потоку через частково відкритий кран.

Величини напруги, струму і опору пов'язані формулою 7=/R, де V – напруга (у вольтах), I – сила струму (в амперах), R – опір (в омах). Іншими словами, напруга на резистивному елементі ланцюга дорівнює добутку струму, що протікає через елемент, на його опір. Це закон Ома, названий на честь Георга Ома, який уперше опублікував цю формулу в 1827 р.

На рис. 2.1 показано просте схематичне представлення цього відношення. Стопка горизонтальних ліній ліворуч позначає джерело напруги, наприклад акумулятор або блок живлення комп'ютера. Зигзагоподібна форма праворуч являє собою резистор. Лінії, що з'єднують компоненти схеми, – це дроти, які вважаються ідеальними провідниками. Ідеальний провідник забезпечує протікання електричного струму без опору.

Струм, позначений буквою I, тече ланцюгом за годинниковою стрілкою, від позитивного полюса батареї, через резистор і назад до негативного полюса батареї. Негативна сторона батареї визначається в цій схемі як опорна точка з нульовою напругою.

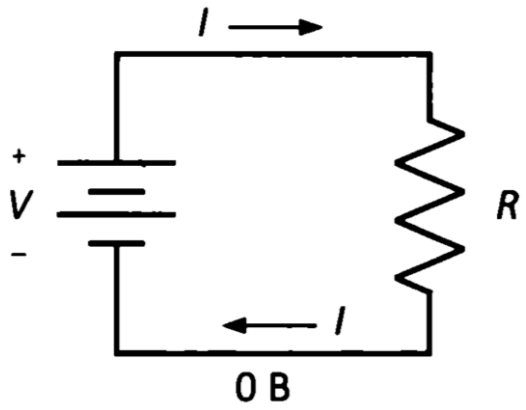


Рис 2.1. *Проста резистивна схема*

За аналогією з водопровідною трубою дріт під нульовою напругою являє собою «басейн із водою». «Насос» (батарея на схемі) забирає воду з басейну і виштовхує її з верхньої частини символу батареї в трубу під вищим тиском. Вода тече у вигляді струму I до крана, позначеного резистором R праворуч. Після проходження через кран, що обмежує потік, вода потрапляє в басейн, де її можна знову набрати насосом.

Якщо ми припустимо, що напруга батареї (тобто підвищення тиску у водяному насосі) постійна, то будь-яке збільшення опору R зменшить струм I на обернено пропорційну величину. Наприклад, подвоєння опору знизить струм удвічі. А подвоєння напруги за рахунок послідовного включення двох батарей, як це роблять у ліхтариках, подвоїть струм через резистор.

У наступному розділі ми познайомимося з транзистором, який слугує основою для всіх сучасних цифрових електронних пристроїв.

Транзистор

Транзистор – це напівпровідниковий пристрій, який, для цілей цього обговорення, функціонує як цифровий перемикач. Напівпровідник – це матеріал, який займає проміжне положення за провідністю між хорошими провідниками (такими як мідний дріт) і хорошими ізоляторами (такими як скло або пластик). За наявності відповідної схеми провідність напівпровідникового пристрою можна змінювати за допомогою керуючого входу. Використовуваний у такий спосіб транзистор стає цифровим перемикальним елементом. Операція перемикання транзистора електрично еквівалентна перемиканню між дуже високим і дуже низьким опором залежно від стану сигналу на керуючому вході. Важлива особливість перемикальних транзисторів полягає в тому, що для керуючого входу не потрібен великий струм.

Це означає, що малий струм на керуючому вході може керувати комутацією набагато більшого струму, що проходить через транзистор. Струм на виході одного транзистора може керувати входами багатьох інших транзисторів. Ця особливість дуже важлива для розроблення складних цифрових схем. На рис. 2.2 показано схематичне позначення транзистора типу n-p-n. Позначення n-p-n відноситься до будови взаємопов'язаних ділянок кремнієвого кристала, що складають транзистор. В ділянку n кристала за допомогою процесу, званого **легуванням**, додається матеріал, що збільшує кількість доступних електронів. Область p легується матеріалом, що зменшує кількість доступних електронів. Транзистор типу n-p-n містить дві області n, між якими розташована область p. Три виводи пристрою приєднані до кожної з цих областей.

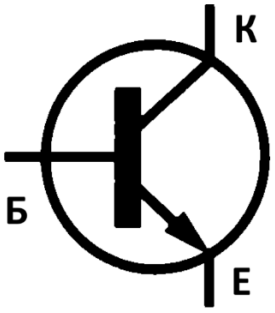


Рис 2.2. *Схематичне позначення транзистора типу n-p-n*

Колектор, позначений літерою К на рис. 2.2, приєднаний до однієї з n областей, а емітер Е – до іншої області n. База Б приєднана до області p між двома областями n. Колектор «збирає» струм, а емітер «випускає» струм у напрямку, зазначеному стрілкою. Вивід бази є керуючим входом. Змінюючи напругу, що подається на вивід бази, і таким чином, змінюючи величину струму, що протікає через базу, можна регулювати струм, що входить через колектор і виходить через емітер.

Логічні вентилі

На рис. 2.3 подано принципову схему транзисторного вентиля НЕ. Живлення схеми забезпечує джерело напругою 5 В. Вхідний сигнал може надходити від схеми з кнопкою, яка видає 5 В, коли кнопку натиснуто, і О В, коли кнопку відпущено. Резистор R_1 обмежує струм, що протікає від клеми входу до виводу бази транзистора, коли на вході високий рівень (близько 5 В). У типовій схемі опір резистора R_1 становить близько 1000 Ом. Резистор R_2 може мати опір 5000 Ом. Він обмежує струм, що протікає від колектора до емітера, коли транзистор увімкнений (відкритий).

На клему входу надходить вхідна напруга в діапазоні від 0 до 5 В, але оскільки ми говоримо про роботу цифрової схеми, нас цікавлять тільки сигнали, близькі до 0 В (низький рівень) або 5 В (високий рівень). Будемо вважати, що всі рівні напруги між низьким і високим є перехідними при майже миттєвих переходах між низьким і високим рівнями.

Типовий n-p-n-транзистор має напругу перемикання близько 0,7 В. Коли на клемі входу підтримується низька напруга, наприклад 0,2 В, транзистор закритий і має дуже великий опір між колектором і емітером. При цьому резистор R_2, приєднаний до джерела живлення 5 В, забезпечує високий рівень сигналу на виході (близько 5 В).

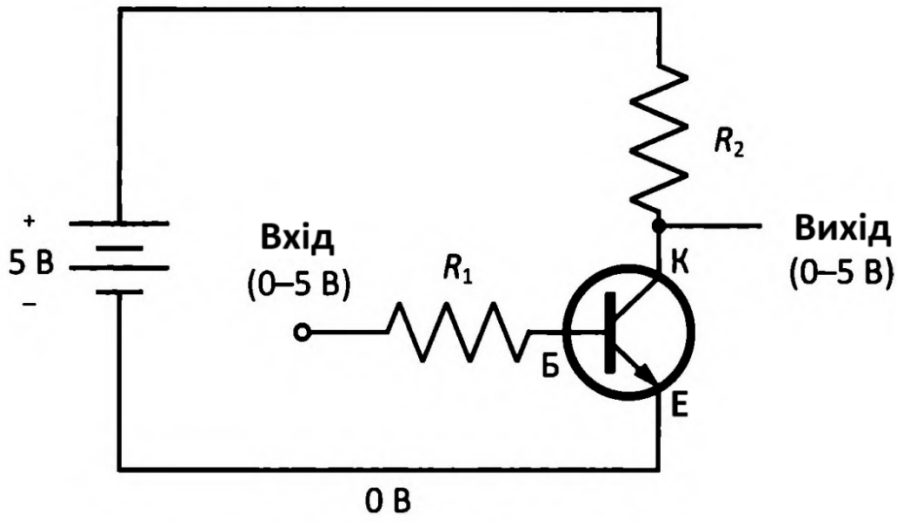


Рис 2.3. *Транзисторний вентиль НЕ*

Коли напруга вхідного сигналу перевищує 0,7 В і досягає значення з діапазону від 2 до 5 В, транзистор відкривається, і опір між колектором і емітером стає дуже маленьким. Тобто, по суті, клема виходу з'єднується з провідником 0 В через ланцюг з опором, який набагато менший, ніж R_2. У результаті на клемі виходу діє низька напруга, зазвичай близько 0,2 В. Загалом поведінку цієї схеми можна виразити так: коли на клемі входу високий рівень, на клемі виходу спостерігається низький рівень. Коли на клемі входу низький рівень, на клемі виходу – високий рівень. Ця функція описує вентиль НЕ, у якому вихід є інверсією входу. Поведінка цього вентиля описана в таблиці істинності, поданій у табл. 2.1 (низькому рівню сигналу відповідає двійкове значення 0, а високому рівню – значення 1).

Таблиця 2.1. *Таблиця істинності вентиля НЕ*

Вихід	**Вхід**
1	0
0	1

Таблиця істинності – це табличне представлення вихідного значення логічного виразу як функції всіх можливих комбінацій вхідних даних. Кожен стовпчик представляє один вхід або вихід, виходи показані в правій частині таблиці. Кожен рядок представляє один набір вхідних значень разом із вихідним значенням виразу з урахуванням цих вхідних значень.

Такі схеми, як вентиль НЕ, показаний на рис. 2.3, настільки поширені в цифровій електроніці, що їм привласнені спеціальні схематичні позначення для побудови діаграм більш високого рівня, що представляють більш складні логічні функції.

Символ вентиля НЕ являє собою трикутник із маленьким кружком на виході (рис. 2.4).

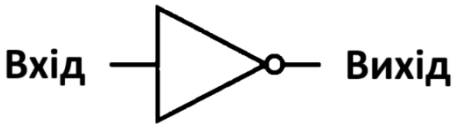


Рис 2.4. *Схематичне позначення вентиля НЕ*

Трикутник позначає підсилювач, вказуючи на те, що цей пристрій перетворює слабкий вхідний сигнал на сильніший вихідний сигнал. Кружок позначає оператор інверсії, який перетворює сигнал на його двійкову протилежність.

Далі ми розглянемо деякі складніші логічні операції, які можна реалізувати, спираючись на схему вентиля НЕ. Схема на рис. 2.5 використовує два транзистори для виконання операції І з двома входами: $Вход_1$ та $Вхід_2$. Операція І дає на виході 1, коли на обох входах 1, в інших випадках вихід дорівнює 0. Резистор R_2 підтримує низький сигнал на виході, якщо високі рівні сигналів на входах $Вход_1$ і $Вхід_2$ не відмикають обидва транзистори.

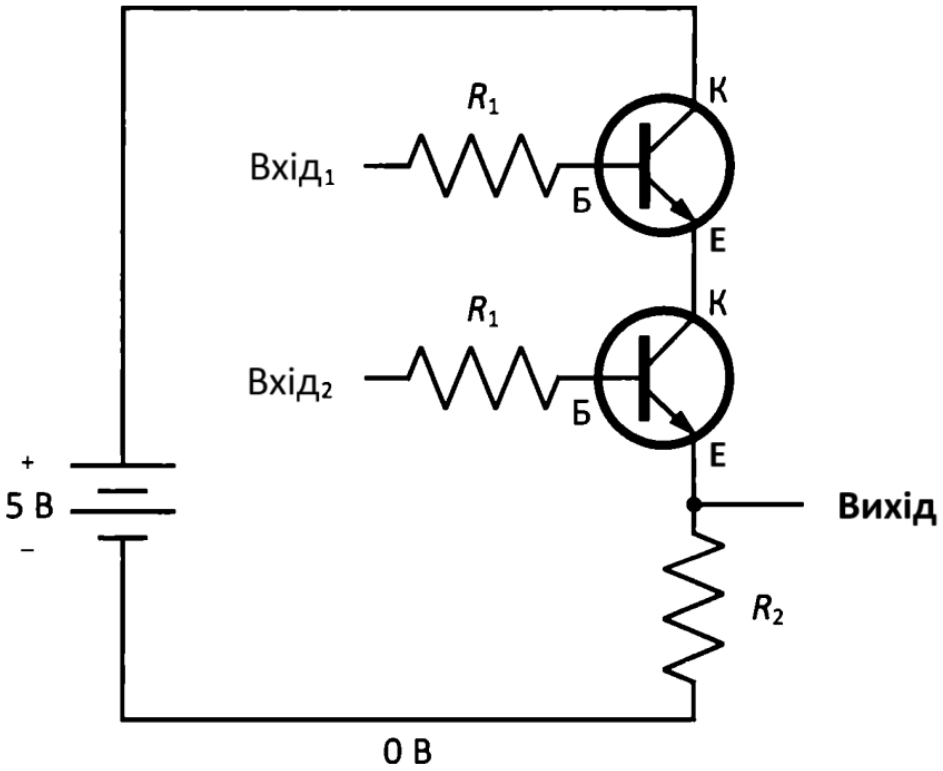


Рис 2.5. *Транзисторний вентиль І*

У табл. 2.2 подано таблицю істинності вентиля І. Простіше кажучи, вихідний сигнал є істинним (має рівень 1), коли обидва входи $Вхід_1$ і $Вхід_2$ є істинними, і хибним (0) в інших випадках.

Таблиця 2.2. *Таблиця істинності вентиля І*

Вхід$_1$	**Вхід$_2$**	**Вихід**
0	0	0
1	0	0
0	1	0
1	1	1

Вентиль І має власне схематичне позначення, показане на рис. 2.6.

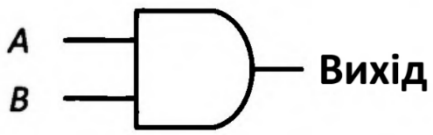


Рис 2.6. *Схематичне позначення вентиля І*

Вентиль АБО видає на виході 1, коли будь-який із входів, А або В, має значення 1, а також коли обидва входи мають значення 1. Таблиця істинності вентиля АБО наведена в табл. 2.3.

Таблиця 2.3. *Таблиця істинності вентиля АБО*

А	**В**	**Вихід**
0	0	0
1	0	1
0	1	1
1	1	1

Схематичне позначення вентиля АБО показано на рис. 2.7.

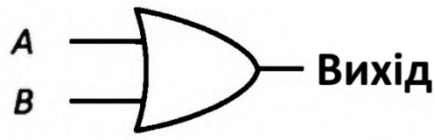


Рис 2.7. *Схематичне позначення вентиля АБО*

Операція виключного АБО видає 1, коли тільки один із входів, А і В, має значення 1. Вихід дорівнює 0, коли обидва входи дорівнюють 0, а також коли обидва входи дорівнюють 1. Таблиця істинності вентиля виключного АБО наведена в табл. 2.4.

Схематичне позначення вентиля виключного АБО показано на рис. 2.8.

Таблиця 2.4. *Таблиця істинності вентиля виключного АБО*

A	B	Вихід
0	0	0
1	0	1
0	1	1
1	1	0

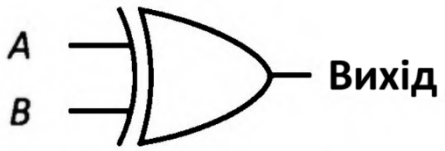


Рис 2.8. *Схематичне позначення вентиля виключного АБО*

Кожен із вентилів – І, АБО і виключне АБО – можна реалізувати з інвертованим виходом. При цьому функція вентиля така сама, як описано вище, за винятком інверсії виходу (у стовпчику «Вихід» табл. 2.2-2.4 нуль замінюється одиницею, а одиниця – нулем). Схематичне позначення вентилів І, АБО або виключного АБО з інвертованим виходом має невелике коло, додане на стороні виходу, як і у випадку з вентилем НЕ. Назви вентилів з інвертованими виходами І-НЕ, АБО-НЕ та виключне АБО-НЕ. НЕ в кожній із цих назв означає НІ. Наприклад, І-НЕ означає інверсію І, що функціонально еквівалентно вентилю І, за яким слідує вентиль НЕ.

Прості логічні вентилі можна комбінувати для отримання більш складних функцій. Мультиплексор – це схема, яка обирає один із кількох входів для передачі на свій вихід залежно від стану входу селектора. На рис. 2.9 показано схему мультиплексора з двома входами.

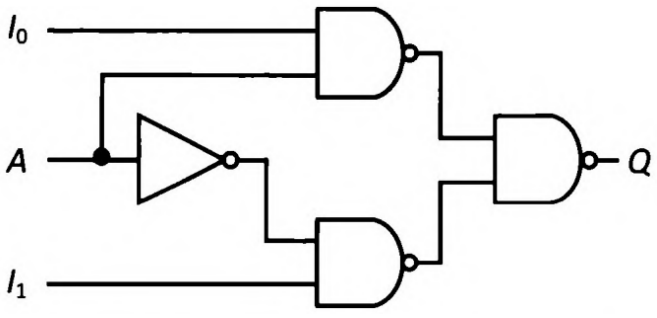


Рис 2.9. *Схема мультиплексора з двома входами*

Схема має два однобітні входи даних: I_0 та I_1. Коли на вході селектора А високий рівень, на вихід Q передають значення I_0. Коли на вході А низький рівень, на вихід передають значення I_1. Одне із застосувань мультиплексора в архітектурі процесора – вибір вхідних даних одного з кількох джерел під час завантаження внутрішнього регістра.

Таблиця істинності мультиплексора з двома входами наведена в табл. 2.5. У цій таблиці значення X означає «байдуже», тобто стан цього сигналу не впливає на визначення стану виходу Q.

Таблиця 2.5. *Таблиця істинності мультиплексора з двома входами*

A	I_0	I_1	Q
1	0	X	0
1	1	X	1
0	X	0	0
0	X	1	1

Логічні вентилі, описані в цьому розділі, і схеми, побудовані на їхній основі, представляють **комбінаційну логіку**, коли значення на виході в будь-який момент часу залежить тільки від поточного стану входів. На даний момент ми ігноруємо затримку поширення і припускаємо, що вихід схеми негайно реагує на зміни входів. Іншими словами, за цих припущень вихід не залежить від попередніх значень входів. Комбінаційні логічні схеми не запам'ятовують попередні значення входів або виходів.

У наступному розділі ми розглянемо деякі схеми, які можуть зберігати відомості про попередні операції.

Засувки

Комбінаційна логіка не дає змоги безпосередньо зберігати дані, що необхідно для реалізації цифрових функцій, таких як регістри процесора. Логічні вентилі можна використовувати для створення елементів зберігання даних шляхом передавання сигналу зворотного зв'язку з виходу одного вентиля на вхід іншого вентиля, який передує першому в ланцюзі передавання сигналу.

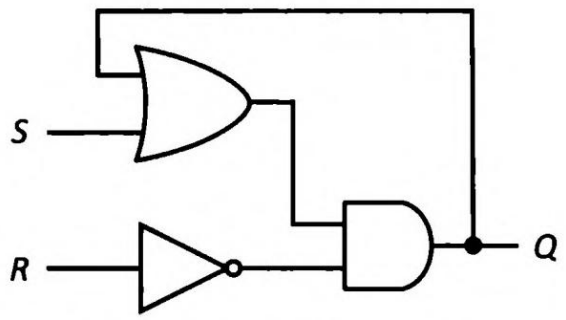


Рис 2.10. *Схема SR-засувки*

Засувка – це однобітний запам'ятовувальний пристрій, побудований із логічних вентилів. На рис. 2.10 показано простий тип засувки, так звана **засувка зі встановленням і скиданням** (set-reset latch) або **SR-засувкою**. Особливістю цієї

схеми, що забезпечує запам’ятовування стану, є зворотний зв’язок з виходу вентиля І на вхід вентиля АБО.

Залежно від стану входів S і R ця схема може встановити вихід Q (перевести його на високий рівень), скинути вихід Q (перевести його на низький рівень) або утримувати на виході Q його останнє значення. У стані утримання обидва входи S і R мають низький рівень, а стан виходу Q зберігається. Подача позитивного імпульсу на вхід S (перехід від низького рівня до високого, а потім повернення на низький рівень) спричиняє перехід виходу Q на високий рівень і утримання цього рівня. Подача позитивного імпульсу на вхід R спричиняє перехід виходу Q на низький рівень і утримання цього рівня. Якщо на обох входах, S і R, діє високий рівень, вхід R має пріоритет над входом S і переводить Q на низький рівень.

Таблиця істинності для RS-засувки наведена в табл. 2.6. $Q_{пред}$ представляє останнє значення виходу Q, вибране в результаті дій входів S і R.

Таблиця 2.6. *Таблиця істинності SR-засувки*

S	R	Дія	Q
0	0	Утримання	$Q_{пред}$
1	0	Встановлення	1
X	1	Скидання	0

Під час роботи з цією схемою засувки і взагалі з пристроями енергозалежної пам’яті слід пам’ятати про те, що початковий стан виходу Q під час увімкнення живлення є невизначеним. Поведінка схеми під час запуску і кінцевий стан виходу Q залежать від характеристик і синхронізації роботи окремих вентилів у міру їхнього спрацьовування. Після ввімкнення живлення і перед початком використання цієї схеми за її призначенням необхідно подати імпульс на вхід S або R, щоб перевести вихід Q у відомий стан.

Керована D-засувка, де **D** означає **дані**, має безліч застосувань у цифрових схемах. Термін **“керована"** вказує на використання додаткового входу, який дозволяє або забороняє проходження даних через схему. На рис. 2.11 показано реалізацію керованої D-засувки.

Стан входу D передається на вихід Q, коли на вході E (дозвіл) високий рівень. Коли на вході E низький рівень, вихід Q зберігає своє попереднє значення незалежно від стану входу O. Вихід Q завжди видає значення, зворотне до виходу O (горизонтальна риска над Q означає НЕ).

Варто приділити деякий час простеженню логічної послідовності цієї схеми, щоб зрозуміти її роботу. Ліва половина рис. 2.11, що складається з входу D, вентиля НЕ і

двох лівих вентилів І-НЕ, являє собою комбінаційну логічну схему, вихід якої завжди є прямою функцією входу.

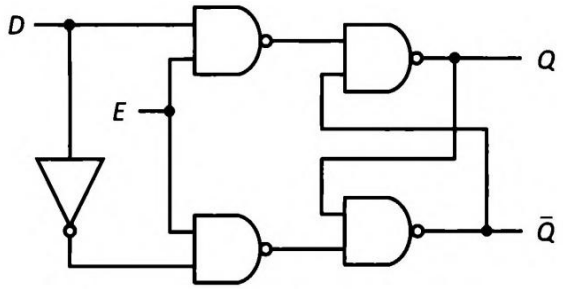


Рис 2.11. *Схема керованої D-засувки*

Таблиця 2.7. *Таблиця істинності керованої D-засувки*

D	E	Q	$\bar{Q}$
0	1	0	1
1	1	1	0
X	0	$Q_{пред}$	$\bar{Q}_{пред}$

Спочатку розглянемо випадок, коли на вході Е діє низький рівень. За низького рівня на вході Е один із входів кожного з двох лівих вентилів І-НЕ має низький рівень, тож вихід обох вентилів дорівнює 1 (див. табл. 2.2 і таблицю істинності вентиля І, не забуваючи, що вентиль І-НЕ еквівалентний вентилю І, за яким іде вентиль НЕ). У цьому стані значення входу D не має значення, і один із виходів, Q або $\bar{Q}$, повинен мати високе значення, а інший – низьке, через перехресне з'єднання виходів двох правих вентилів І-НЕ з їхніми входами. Цей стан зберігатиметься доти, доки на вході Е діє низький рівень.

Коли на вході Е високий рівень, то, залежно від стану входу Я, на виході одного з двох лівих вентилів І-НЕ буде низький рівень, а на виході іншого – високий. Той із них, на виході якого діє низьке значення, забезпечить високий вихідний сигнал на приєднаному до нього правому вентилі І-НЕ. Цей вихідний сигнал подається у вигляді зворотного зв'язку на вхід іншого правого вентиля І-НЕ і за високого рівня на обох входах на виході буде низький рівень. У результаті вхідний сигнал О передається на вихід Q, а його зворотне значення з'являється на виході $\bar{Q}$.

Важливо розуміти, що значення на виходах Q і $\bar{Q}$ не можуть бути одночасно високими або низькими, тому що така ситуація є конфліктом між виходами і входами двох правих вентилів І-НЕ. Якщо одна з цих умов виникне випадково, наприклад під час подавання живлення, схема автоматично налаштується на стабільну конфігурацію, за якої Q і $\bar{Q}$ перебуватимуть у протилежних станах. Як і у випадку з SR-засувки, результат цієї автоматичної корекції непередбачуваний, тому, перш ніж

використовувати керовану D-засувку у будь-яких операціях, необхідно перевести її у відомий початковий стан.

Ініціалізацію виконують шляхом встановлення високого рівня на вході E, вибору за допомогою входу O бажаного початкового значення на виході Q і потім встановлення низького рівня на вході E.

Описана вище керована D-засувка є **пристроєм, що запускається за рівнем сигналу**, тобто значення на виході Q змінюється залежно від значення на вході O, поки на вході E підтримується високий рівень. У складніших цифрових схемах важливим завданням стає синхронізація кількох послідовно з'єднаних елементів схеми, що дає змогу позбутися необхідності ретельно враховувати затримки поширення сигналу між окремими пристроями. Використання **загального тактового сигналу** як вхідного сигналу для кількох елементів забезпечує таку синхронізацію.

У конфігурації зі спільним тактовим генератором компоненти оновлюють стани своїх виходів у момент проходження фронту або спаду імпульсу тактового сигналу (тобто в момент переходу від низького рівня до високого або від високого рівня до низького), а не безперервно відстежують високий або низький рівень вхідного сигналу.

Спрацьовування за фронтом або спадом імпульсів тактового сигналу корисне, оскільки ці події визначають точні моменти, у які входи пристрою мають бути стабільними та достовірними. Після проходження фронту або спаду тактового сигналу входи пристрою можуть довільно змінюватися під час підготовки до наступного активного фронту або спаду тактового сигналу, не впливаючи на рівень сигналу на виходах схеми. Розглянута далі схема тригера реагує на фронти тактових імпульсів, забезпечуючи цю бажану характеристику для складних цифрових схем.

Тригери

Пристрій, який змінює стан виходу тільки тоді, коли тактовий сигнал здійснює заданий перехід (від низького рівня до високого або від високого рівня до низького), називається **пристроєм, що запускається за фронтом сигналу**. **Тригери** схожі на засувки, з тією лише різницею, що сигнал на виході тригера визначається впливом фронту тактового сигналу, а не безперервного вхідного сигналу протягом терміну його дії.

D-тригер, що спрацьовує за фронтом, є популярним компонентом цифрових схем і застосовується для розв'язання найрізноманітніших завдань. У ньому зазвичай використовуються вхідні сигнали встановлення і скидання, які виконують ті самі функції, що й у SR-засувки. Цей тригер має вхід D, який функціонує так само, як вхід D керованої D-засувки. Замість входу дозволу в D-тригера є тактовий вхід, який запускає передавання сигналу з входу D на вихід Q і, з іншою версією, на вихід $\bar{Q}$ за фронтом тактового сигналу. Тригер не реагує на зміну стану входу D за межами дуже вузького часового вікна, що оточує фронт тактового сигналу. Зміни стану

входів S і R мають пріоритет над будь-якими змінами на вході D і вході тактового сигналу.

На рис. 2.12 представлено схематичне позначення D-тригера. Вхід тактового сигналу позначено маленьким трикутником у лівій частині символу.

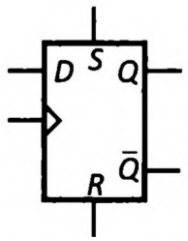


Рис 2.12. *Схематичне позначення D-тригера*

Розглянемо табл. 2.8. Спрямовані вгору стрілки в стовпчику CLK вказують на фронт тактового сигналу. Значення для виходів Q і $\bar{Q}$ у рядках таблиці зі спрямованими вгору стрілками представляють стан виходів після проходження фронту тактового сигналу.

Таблиця 2.8. *Таблиця істинності керованої D-засувки*

S	R	D	CLK	Q	$\bar{Q}$
0	1	1	↑	0	1
1	1	0	↑	1	0
X	0	X	Стаб.	$Q_{пред}$	$\bar{Q}_{пред}$
1	0	X	X	1	0
0	1	X	X	0	1

Тригери можна з’єднати послідовно, щоб забезпечити передачу бітів даних від одного тригера до іншого в послідовних тактових циклах. Це досягається шляхом приєднання виходу Q першого тригера до входу D другого тригера з аналогічним продовженням для будь-якої кількості ступенів. Ця структура, звана **регістром зсуву**, має безліч варіантів застосування, включно з послідовно-паралельним перетворенням і паралельно-послідовним перетворенням.

Якщо вихід Q у кінці регістра зсуву під’єднати до входу D на іншому кінці регістра, вийде **кільцевий лічильник**. Кільцеві лічильники використовують для таких завдань, як побудова **кінцевих автоматів**. Останні реалізують математичну модель, яка завжди перебуває в одному з низки чітко визначених станів. Переходи між станами відбуваються, коли значення на входах задовольняють вимогам для переходу в інший стан.

Кільцевий лічильник, представлений на рис. 2.13, має чотири позиції. Лічильник ініціалізується шляхом подання на вхід RST позитивного імпульсу, тобто спочатку високого, а потім низького рівня. Цей імпульс встановлює вихід Q першого (крайнього лівого) тригера в стан 1, а виходи Q інших тригерів – в стан 0.

Після цього кожен фронт сигналу на вході CLK передає 1 біт у наступний за порядком тригер. Четвертий імпульс CLK передає 1 назад у крайній лівий тригер. У будь який момент часу на виході Q одного тригера – 1, а виходи всіх інших тригерів мають значення 0.

Тригери – це пристрої, що запускаються за фронтом імпульсу, і всі вони керуються загальним тактовим сигналом, що робить цю схему **синхронною**.

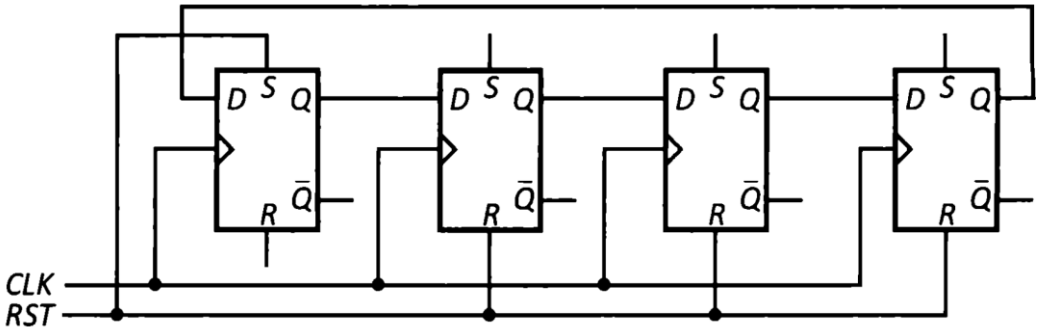


Рис 2.13. *Схема чотирипозиційного кільцевого лічильника*

Ця схема містить чотири стани кільцевого лічильника. Якщо додати ще шість тригерів, то кількість станів сягне 10. Як ми вже говорили в *розділі* 1, у машині ENIAC для збереження стану десяткових цифр використовувалися 10-позиційні кільцеві лічильники на основі електронних ламп. Кільцевий лічильник, що зберігає 10 станів, заснований на схемі, наведеній на рис. 2.13, може виконувати ту саму функцію.

У наступному розділі ми створимо з тригерів регістри для зберігання даних.

Регістри

Регістри процесора тимчасово зберігають значення даних і слугують місцем зберігання вхідних і вихідних даних для операцій різноманітних інструкцій, включно з переміщенням даних у пам’ять і з пам’яті, арифметичними операціями та маніпуляціями з бітами. У більшості процесорів загального призначення передбачено інструкції для зсуву двійкових значень, що зберігаються в регістрах, вліво або вправо, а також для виконання операцій обертання, під час яких біти даних, зміщені з одного кінця регістра, вставляються з протилежного кінця. Операція обертання аналогічна кільцевому лічильнику, за винятком того, що біти в циклі можуть містити довільні значення, тоді як кільцевий лічильник зазвичай передає один біт, що дорівнює одиниці, через послідовність позицій. Схеми, що виконують ці функції, будуються з низькорівневих вентилів і тригерів, представлених раніше в цьому розділі.

Операції запису і читання даних регістрів усередині процесора зазвичай виконуються паралельно. Це означає, що всі біти записуються або зчитуються за окремими сигнальними лініями одночасно під керуванням фронту загального тактового сигналу. У прикладах, наведених у цьому розділі, для простоти використовуються 4-розрядні регістри, але ці схеми легко розширити до 8, 16, 32 або 64 розрядів.

На рис. 2.14 показано простий 4-розрядний регістр з паралельними введенням і виведенням. Це синхронна схема, у якій біти даних, що надходять на входи $D_0 – D_3$, завантажуються в тригери по фронту тактового сигналу CLK. Біти даних одразу ж з'являються на виходах $Q_0 – Q_3$ і зберігають свій стан доти, доки нові значення даних не будуть завантажені по фронту наступного імпульсу тактового сигналу.

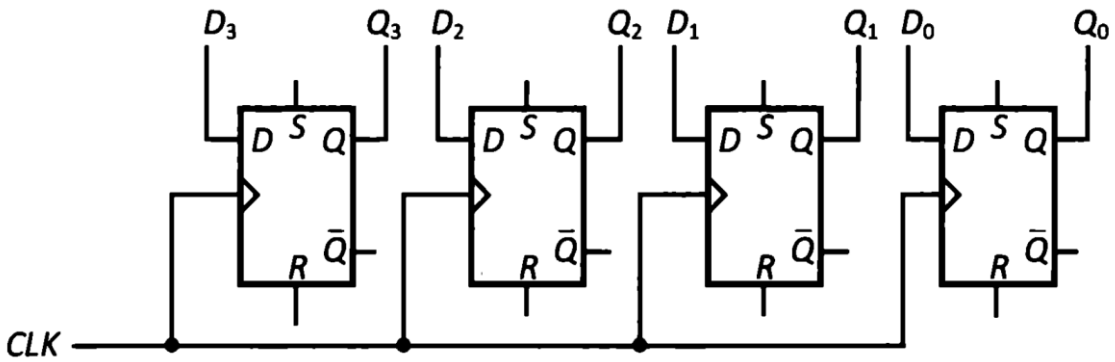


Рис 2.14. *Схема 4-розрядного регістра*

Для реалізації корисних функцій, крім простого зберігання даних у регістрі, необхідна можливість завантажувати в регістр дані з декількох джерел, виконувати операції з вмістом регістра і записувати отримані значення в одне з місць призначення, яких може бути кілька.

У процесорах загального призначення дані зазвичай завантажуються в регістр із комірки пам'яті, через вхідний порт або шляхом перенесення з іншого регістра. Операції, що виконуються над вмістом регістра, можуть містити збільшення на 1, зменшення на 1, арифметичні операції, зсув, обертання та бітові маніпулювання, такі як операції І, АБО та виключаюче АБО. Зверніть увагу, що збільшення або зменшення цілого числа на 1 еквівалентно операції додавання або віднімання одного операнда з другим мається на увазі операндом, що дорівнює 1. Після внесення в регістр результату обчислення його вміст може бути записано в комірку пам'яті, у вихідний порт або в інший регістр.

На рис. 2.9 показано схему мультиплексора з двома входами. Цю схему легко розширити, щоб вона підтримувала більшу кількість входів, будь-який з яких може бути обраний керуючими сигналами. Однобітовий мультиплексор можна реплікувати для підтримки одночасної роботи з усіма бітами слова процесора. Така схема використовується для вибору серед безлічі джерел під час завантаження даних у регістр. Під час реалізації в процесорі логіка, що запускається опціями кодів інструкцій, встановлює керуючі входи мультиплексора для маршрутизації даних від обраного джерела в зазначений регістр призначення. У розділі 3 використання мультиплексорів буде розширено для маршрутизації даних до регістрів та інших блоків усередині процесора.

У наступному розділі будуть представлені схеми для додавання двійкових чисел.

Суматори

Процесори загального призначення зазвичай підтримують операцію додавання для виконання обчислень над значеннями даних і окремо для керування покажчиком інструкцій. Після виконання кожної інструкції покажчик інструкцій збільшується на 1 для вказівки на позицію наступної інструкції.

Якщо процесор підтримує інструкції, що складаються з декількох слів, нове значення покажчика інструкцій має дорівнювати своєму поточному значенню плюс кількість слів у щойно виконаній інструкції.

Схема простого суматора додає два біти даних, враховує перенесення з попереднього розряду і видає на виході однобітну суму та ознаку перенесення. Ця схема, показана на рис. 2.15, називається **повним суматором**, оскільки вона враховує вхідне перенесення. **Напівсуматор** складає тільки два біти даних без урахування перенесення з попереднього розряду.

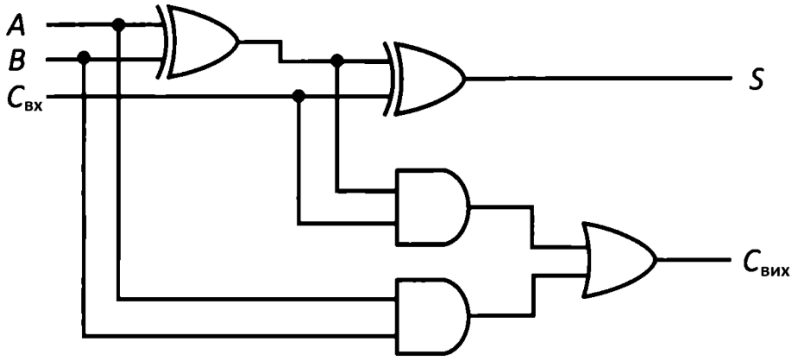


Рис 2.15. *Схема повного суматора*

Повний суматор використовує логічні вентилі для отримання результату на виході в такий спосіб. Біт суми S дорівнює 1 тільки в тому випадку, якщо загальна кількість бітів зі значенням 1 у наборі A, B, $C_{вх}$ є непарним числом. В іншому випадку біт S дорівнює 0. Цю логічну операцію виконують два вентиля типу "виключне АБО". Вихід $C_{вих}$ дорівнює 1, якщо обидва входи, A і B, дорівнюють 1, або якщо тільки один із входів, A або B, дорівнює 1, і $C_{вх}$ також дорівнює 1. Інакше $C_{вих}$ дорівнює 0.

Для використання в схемах вищого рівня схему, показану на рис. 2.15, можна звести до схематичного блоку, має три входи та два виходи. На рис. 2.16 показано 4-розрядний суматор із чотирма блоками, що представляють копії схеми повного суматора з рис. 2.15. Вхідні дані – це два слова, що їх треба скласти, $A_0 – A_3$ і $B_0 – B_3$, та ознака перенесення з попереднього розряду $C_{вх}$. Результатом будуть сума $S_0 – S_3$ і перенесення в наступний розряд $C_{вих}$.

Важливо зазначити, що ця схема є комбінаційною. Це означає, що після встановлення значень на входах вихідні значення формуються безпосередньо. Це також стосується і поширення від біта до біта, незалежно від того, на яку кількість бітів впливають

перенесення. Оскільки перенесення здійснюється по бітах, ця конфігурація називається **суматором з наскрізним перенесенням**.

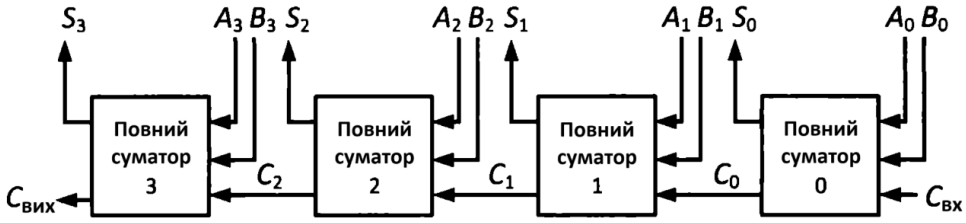


Рис 2.16. *Схема 4-розрядного суматора*

Потрібен певний час, щоб переноси поширилися по всіх позиціях бітів і на виходах стабілізувалися кінцеві значення.

Оскільки ми зараз обговорюємо схему, в якій шлях сигналу проходить через значну кількість пристроїв, доречно обговорити чинники, що впливають на час, необхідний для проходження сигналів через безліч компонентів.

Затримка поширення

При зміні вхідного сигналу логічного пристрою його вихідний сигнал змінюється не миттєво. Між зміною стану на вході і появою результату на виході є тимчасова затримка. Вона називається **затримкою поширення**. Затримка поширення сигналу через схему встановлює верхню межу тактової частоти, на якій може працювати схема. У мікропроцесорі тактова частота визначає швидкість, з якою пристрій може виконувати інструкції.

Послідовне з'єднання кількох комбінаційних схем призводить до загальної затримки поширення, що дорівнює сумі затримок окремих пристроїв. Вентиль може мати різні значення затримки поширення сигналу для переходу від низького рівня до високого і для переходу від високого рівня до низького, тому під час оцінювання найгіршої затримки в контурі слід використовувати більше з цих двох значень.

Як показано на рис. 2.15, найдовший шлях (з погляду кількості послідовно ввімкнутих вентилів) від входу до виходу для повного суматора – від входів A і B до виходу $C_{вих}$: всього три послідовних вентиля. Якщо всі вхідні сигнали 4-розрядного суматора на рис. 2.16 встановлюються одночасно, замикання в трьох вентилях, пов'язане із входами A і B, відбуватиметься одночасно в усіх чотирьох суматорах. Однак вихід C_0 повного суматора 0 буде гарантовано стабільним тільки після затримки в трьох вентилях цього суматора. Після того як C_0 стабілізується, виникає додаткова затримка проходження сигналу через два вентилі повного суматора 1 (зверніть увагу на рис. 2.15, де видно, що сигнал із $C_{вх}$, проходить лише через два послідовні рівні вентилів).

Таким чином, загальна затримка поширення для схеми на рис. 2.16 становить три затримки у вентилях на повному суматорі 0, за якими слідують по дві затримки у

вентилях на кожному з інших трьох повних суматорів – загалом дев'ять затримок у вентилях. Така затримка може здатися не надто значною, але розглянемо 32-розрядний суматор: загальна затримка розповсюдження для цього суматора становить три затримки у вентилях для повного суматора 0 плюс дві затримки кожного з інших суматорів – загалом 65 затримок у вентилях.

Шлях із максимальною затримкою поширення сигналу через комбінаційну схему називається **критичним шляхом**. Затримка критичного шляху встановлює верхню межу тактової частоти, яку можна використовувати для керування схемою.

Сімейство логічних вентилів на **вдосконалених транзисторно-транзисторних логічних схемах із бар'єрами Шоткі** (advanced Schottky transistor-transistor logic, (AS) TTL), які позначають скороченням **УТТЛШ**, є одним із найшвидших вентилів в окремому корпусі, доступних на сьогоднішній день.

За типового навантаження УТТЛШ-вентиль І-НЕ має затримку поширення **2 наносекунди (нс)**. Для порівняння: за 2 нс світло у вакуумі проходить трохи менше 60 см.

У 32-розрядному суматорі з наскрізним перенесенням 65 затримок поширення сигналу через УТТЛШ-вентилі складаються в загальну затримку між встановленням вхідних сигналів і отриманням стабільних кінцевих вихідних значень, що становить 130 нс. Для того щоб зробити приблизну оцінку, припустимо, що це найбільша можлива затримка поширення сигналу через усю інтегральну схему процесора. Ми також проігноруємо будь-який додатковий час, для збереження стабільності входів до і після активного фронту тактового сигналу. Таким чином, цей суматор не може виконувати послідовні операції з вхідними даними частіше, ніж раз на 130 нс.

При виконанні 32-розрядного додавання за допомогою суматора з наскрізним перенесенням процесор використовує фронт тактового сигналу для перенесення вмісту двох регістрів (кожний з яких складається з набору D-тригерів) і прапора C процесора на входи суматора. Наступний фронт тактового сигналу викликає навантаження результатів додавання в регістр призначення. Прапор C процесора приймає значення $C_{вих}$ від суматора.

Тактовий сигнал з періодом 130 нс має частоту 1/130 нс, що становить 7,6 МГц. Ця частота, звісно, не виглядає надто високою, особливо якщо врахувати, що сьогодні доступно багато недорогих процесорів із тактовою частотою понад 4 ГГц. Однією з причин цього розбіжності є властива інтегральним схемам, що містять велику кількість щільно скомпонованих транзисторів, перевага у швидкодії, а друга зумовлена винахідливістю розробників, про що згадував Гордон Мур (див. главу 1). Для ефективного виконання функцій суматора було розроблено безліч оптимізованих рішень, що дають змогу істотно зменшити найбільшу затримку поширення. У *розділі 8* ми обговоримо деякі методи, які використовують архітектори процесорів, щоб домогтися більш високих швидкостей у своїх конетрукціях.

На додаток до затримок вентилів існує також деяка затримка, що виникає в результаті проходження сигналу по проводах і струмоведучих доріжках інтегральної схеми. Швидкість проходження сигналу по проводах або через інший тип провідного матеріалу змінюється залежно від властивостей матеріалу провідника та ізоляційного матеріалу, що оточує провідник. Залежно від цих та інших факторів швидкість проходження сигналу в цифрових схемах зазвичай становить 50–90% швидкості світла у вакуумі.

У наступному розділі обговорюється генерування і використання тактових сигналів у цифрових схемах.

Синхронізація

Тактовий сигнал задає "серцевий ритм" процесора. Цей сигнал зазвичай має прямокутну форму і фіксовану частоту. Прямокутний сигнал – це цифровий сигнал, що поперемінно набуває високого та низького значень, при цьому в кожному циклі перебування сигналу на високому та низькому рівнях займає рівні проміжки часу. На рис. 2.17 показано приклад зміни прямокутного сигналу в часі.

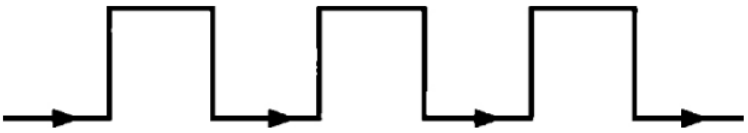


Рис 2.17. *Прямокутний сигнал*

Тактовий сигнал у комп’ютерній системі зазвичай генерує кварцовий генератор із базовою частотою в кілька **мегагерц (МГц)**. 1 МГц – це 1 млн циклів на секунду. Принцип дії кварцового генератора ґрунтується на використанні резонансних коливань фізичного кристала (зазвичай виготовленого з кварцу) для формування циклічного електричного сигналу. Механічна вібрація кристала перетворюється в електричний сигнал за допомогою п’єзоелектричного ефекту. **П’єзоелектричний ефект** – це накопичення електричного заряду в певних кристалах під час механічного впливу. Кристали кварцу резонують на точних частотах, тому їх використовують як елементи синхронізації в комп’ютерах, наручних годинниках та інших цифрових пристроях.

Кварцові генератори є більш точними джерелами тактових сигналів синхронізація, ніж альтернативні рішення, що знаходять застосування в недорогих пристроях, однак кристалам властиво помилки в частоті, що накопичуються протягом кількох днів і тижнів, поступово доводячи відхилення від правильного часу до секунд, а потім – до хвилин. Щоб уникнути цієї проблеми, більшість комп’ютерів, під’єднаних до Інтернету, періодично звертаються до сервера часу, щоб привести свій внутрішній годинник до поточного часу, який публікує точний атомний еталонний годинник.

Схеми помножувача частоти на основі **контуру фазового автоматичного підстроювання частоти** (ФАПЧ) використовують для генерування високочастотних тактових сигналів, необхідних процесорам, що працюють на частотах у кілька гігагерц. Помножувач частоти з ФАПЧ генерує вихідний сигнал прямокутної форми з певною частотою, що виражається цілим числом, кратним вхідній , яка подається на нього від кварцового генератора. Відношення частоти тактового сигналу на виході помножувача з ФАПЧ до частоти його вхідного сигналу називається **множником тактової частоти**.

Множник частоти з ФАПЧ працює шляхом безперервного регулювання частоти свого внутрішнього генератора для підтримання необхідного множника тактової частоти відносно вхідної частоти ФАПЧ. Сучасні процесори зазвичай мають вхід тактового сигналу кварцового генератора і містять кілька розумних жильців частоти з ФАПЧ, що генерують різні частоти. Ці вихідні частоти ФАПЧ потім керують основними операціями процесора з максимально можливою швидкістю, водночас взаємодіючи з компонентами, що потребують більш низьких тактових частот, такими як системна пам’ять і периферійні пристрої.

Послідовна логіка

Цифрові схеми, що генерують вихідні сигнали на основі комбінації поточних і колишніх станів входів, називаються **послідовними логічними схемами**. Цим вона відрізняється від комбінаційної логіки, де стани входів залежать тільки від поточного стану входів.

Якщо послідовна логічна схема, що складається з декількох компонентів, керує цими компонентами за допомогою загального тактового сигналу, ця схема реалізує синхронну логіку.

Дії, пов’язані з виконанням інструкцій процесора, являють собою серію дискретних операцій, що приймають вхідні дані у вигляді опкодів інструкцій і значень даних, одержуваних із різних джерел. Ця діяльність здійснюється при координації на основі головного тактового сигналу. Процесор зберігає інформацію про внутрішній стан від одного такту до наступного і від однієї інструкції до іншої.

Сучасні складні цифрові пристрої, включно з процесорами, майже завжди реалізуються як синхронні послідовні логічні пристрої. Внутрішні компоненти низького рівня, такі як вентилі, мультиплексори, регістри та суматори, розглянуті раніше, зазвичай являють собою комбінаційні логічні схеми. Ці низькорівневі компоненти, своєю чергою, отримують вхідні дані під керуванням синхронної логіки. Після достатнього часу для поширення сигналу комбінаційними компонентами загальний тактовий сигнал передає вихідні дані цих компонентів іншим частинам архітектури під керуванням інструкцій процесора і логічних схем, які виконують ці інструкції.

У *розділі 3* буде представлено компоненти процесора вищого рівня, що реалізують складніші функціональні можливості, включно з декодуванням інструкцій, їх виконанням та арифметичними операціями.

Мови опису апаратних засобів

Прості цифрові схеми легко уявити за допомогою логічних схем, подібних до тих, які були наведені раніше в цьому розділі. Однак під час проектування складніших цифрових пристроїв робота з логічними схемами швидко стає громіздким і незручним рішенням. Як альтернативу логічним схемам за минулі роки було розроблено кілька **мов опису апаратних засобів**. Цьому розробленню сприяв **закон Мура**, який спонукає проєктувальників цифрових систем постійно знаходити нові способи найшвидшого й найефективнішого використання постійно зростаючої кількості транзисторів, доступних в інтегральних схемах.

Мови опису апаратних засобів не є виключною прерогативою розробників цифрових схем у компаніях-виробниках напівпровідників; так само аматори можуть придбати і використовувати ці потужні інструменти за доступною ціною. Деякі з них надаються безкоштовно.

Вентильна матриця – це логічний пристрій, що містить безліч логічних елементів, як-от вентилі І-НЕ і D-тригери, які можна з'єднати один з одним для формування довільних схем.

За допомогою вентильних матриць, віднесених до категорії **програмованих користувачем вентильних матриць** (ПКВМ), кінцеві користувачі можуть реагувати на власні проєкти, використовуючи тільки комп'ютер, невелику макетну плату і відповідний пакет програмного забезпечення.

Розробник може спроектувати складну цифрову схему за допомогою мови опису апаратних засобів і запрограмувати її безпосередньо в мікросхему, унаслідок чого вийде повнофункціональний і високопродуктивний користувацький цифровий пристрій. Сучасні недорогі ПКВМ містять достатню кількість вентилів для реалізації складних сучасних процесорних архітектур. Як приклад можна навести запрограмовану на ПКВМ архітектуру процесора RISC-V (докладно обговорюється в *розділі 11*), доступну у вигляді коду на мові опису апаратних засобів з відкритим вихідним кодом.

VHDL

VHDL – це одна з найпоширеніших на сьогоднішній день мов опису апаратних засобів. Розроблення VHDL розпочалося в 1983 р. на замовлення Міністерства оборони США. Синтаксис і частина семантики VHDL засновані на **мові програмування Ada**. До речі, мова Ada отримала свою назву на честь Ади Лавлейс, програміста аналітичної машини Чарльза Беббіджа, яка обговорюється *в розділі 1*.

Іншою популярною мовою опису апаратних засобів із можливостями, аналогічними VHDL, є Verilog. У цій книзі використовуватиметься виключно VHDL, але наведені тут приклади також можна легко реалізувати мовою **Verilog**.

VHDL – це багаторівнева абревіатура, у якій «V» розшифровується як **VHSIC** (very high-speed integrated circuit), що означає **надшвидка інтегральна схема** (НШІС), а **VHDL** (VHSIC hardware description language) означає **мову опису апаратних засобів НШІС**. Наведений нижче код являє собою реалізацію на VHDL схеми повного суматора, показаної на рис. 2.15.

```
-- Завантаження стандартних бібліотек

library IEEE;
  use IEEE.STD_L0GIC_1164.ALL;

-- Визначення входів і виходів повного суматора

entity FULL_ADDER is
  port (
   A     : in   std_logic;
   B     : in   std_logic;
   C_IN  : in   std_logic;
   S     : out  std_logic;
   C_OUT : out  std_logic
  );
end entity FULL_ADDER;

-- Визначення поведінки повного суматора

architecture BEHAVIORAL of FULL_ADDER is

begin

  S     <= (A XOR B) XOR C_IN;
  C_OUT <= (A AND B) OR ((A XOR B) AND C_IN);

end architecture BEHAVIORAL;
```

Тут розділ, що починається з entity `FULL_ADDER`, визначає входи і виходи повного суматора. Розділ architecture наприкінці коду описує, як працює логіка схеми для отримання вихідних сигналів S і C OUT на основі вхідних сигналів *A, B* і *C_IN*. Термін

`std_logic` вказує на однобітовий двійковий тип даних. Символи <= відображають призначення сигналу, подібне до дротового з'єднання, що передає на вихід, зазначений у лівій частині, значення, обчислене в правій частині.

Наступний код посилається на VDHL-опис повного суматора `FULL_ADDER`, як на компонент реалізації 4-розрядного суматора, представленого на рис. 2.16.

```
-- Завантаження стандартних бібліотек


library IEEE;

  use IEEE.STD_L0GIC_1164.ALL;


-- Визначення входів і виходів 4-розрядного суматора


entity ADDER4 is

  port (

   A4      : in std_logic_vector(3 downto 0);

   B4      : in std_logic_vector(3 downto 0);

   SUM4    : out std_logic_vector(3 downto 0);

   C_OUT4  : out std_logic

  );

end entity ADDER4;


-- Визначення поведінки 4-розрядного суматора


architecture BEHAVIORAL of ADDER4 is


-- Посилання на попереднє визначення повного суматора
```

```
  component FULL_ADDER is
   port (
    A      : in std_logic;
    B      : in std_logic;
    C_IN   : in std_logic;
    S      :out std_logic;
    C_OUT  : out std_logic
  );
  end component;

-- Визначення внутрішніх сигналів 4-розрядного суматора
  signal c0, cl, c2 : std_logic;

begin

-- Призначення входу переносу першого суматора значення 0
  FULL_ADDER0 : FULL_ADDER
   port map (
    A      => A4(0),
    B      => B4(0),
    C_IN   => '0',
    S      => SUM4(0),
    C_OUT  => c0
   );
```

```
FULL ADDER1. : FULL ADDER
  port map (
   A      => A4(l),
   B      => B4(l),
   C_IN   => c0,
   S      => SUM4(1),
   C_OUT  => cl
  );

FULL ADDER2 : FULL ADDER
  port map (
   A      => A4(2),
   B      => B4(2),
   C_IN   => cl,
   S      => SUM4(2),
   C_OUT  => c2
  );

FULL ADDER3; : FULL ADDER
  port map (
   A      => A4(3),
   B      => B4(3),
   C_IN   => c2,
   S      => SUM4(3),
   C OUT  => C_OUT4
```

```
    );


  end architecture BEHAVIORAL;
```

Тут фрагмент коду, що починається з entity ADDER4, визначає входи і виходи 4-розрядного суматора. Фраза `std_logic_vector(3 downto 0)` представляє тип даних у вигляді 4-розрядного вектора з бітом № 3 у найбільш значущій позиції та бітом № 0 у найменш значущій позиції.

Компонент `full_adder` визначено в окремому файлі, посилання на який наведено на початку розділу `component FULL_ADDER is`. Вираз `signal c0, c1, c2 : std_logic;` визначає внутрішні значення переносу між повними суматорами. Чотири розділи `port map` визначають з'єднання між сигналами 4-розрядного суматора і входами/виходами кожного з одноразрядних повних суматорів. Для посилання на біт у бітовому векторі номер біта слідує за ім'ям параметра в круглих дужках. Наприклад, `A4(0)` посилається на найменш значущий із чотирьох бітів в `A4`.

Зверніть увагу на використання ієрархії в цій структурі. Спочатку в окремому, автономному блоці коду було визначено простий компонент – одноразовий повний суматор. Потім цей блок було використано для побудови складнішої схеми – 4-розрядного суматора. Такий ієрархічний підхід можна поширити на багато рівнів, щоб визначити складний цифровий пристрій, побудований з менш складних компонентів, кожен з яких, своєю чергою, складається з ще простіших частин. Цей загальний підхід зазвичай використовують під час розроблення сучасних процесорів, що містять мільярди транзисторів, і дає змогу контролювати рівень складності, щоб архітектура була зрозуміла людям на кожному рівні ієрархії.

Код, представлений у цьому розділі, містить усі схемні визначення, які потрібні програмному інструменту логічного синтезу для реалізації 4-розрядного суматора як компонента ПКВМ. Звичайно, необхідна додаткова схема для передання в схему суматора осмислених вхідних даних, а також для оброблення результатів операції додавання з урахуванням затримки проходження сигналу.

У цьому розділі представлено досить короткий вступ до VHDL. Мета полягає в тому, щоб дати вам зрозуміти, що мови опису апаратних засобів, такі як VHDL, є сучасним досягненням у сфері проєктування складних цифрових схем. Крім того, вам слід знати, що є деякі зовсім недорогі інструменти та пристрої для розробки ПКВМ. У вправах, наведених наприкінці цього розділу, ви познайомитеся з деякими корисними безкоштовними інструментами розробки ПКВМ. Рекомендую вам пошукати інформацію в Інтернеті, щоб дізнатися більше про VHDL та інші мови опису апаратних засобів, а також спробувати свої сили в розробленні деяких простих (і не дуже простих) схем.

Резюме

Цей розділ розпочався з введення у властивості електричних ланцюгів, яке показало, як компоненти, такі як джерела напруги, резистори та дроти, зображуються на схемах. Було представлено транзистор з акцентом на його використанні як елемента перемикання в цифрових схемах. Із транзисторів і резисторів було складено схеми вентилів НЕ та І. Було визначено логічні елементи інших типів, і для кожного пристрою подано таблиці істинності. Логічні елементи використовували для побудови складніших цифрових схем, включно із засувками, тригерами, регістрами та суматорами. Було запроваджено концепцію послідовної логіки та обговорено її застосовність для проектування процесорів. Нарешті, були представлені мови опису апаратних засобів і приклад опису 4-розрядного суматора мовою VHDL.

Тепер ви знайомі з основними концепціями цифрових схем та інструментами проектування, що використовуються під час розроблення сучасних процесорів. У наступному розділі ми детальніше розглянемо ці структурні елементи, щоб вивчити функціональні складові сучасних процесорів, а також обговоримо, як координується робота цих компонентів для реалізації основного робочого циклу процесора, який складається із завантаження, декодування та виконання інструкцій.

Вправи

1. Змініть схему, наведену на рис. 2.5, щоб перетворити вентиль І на вентиль І-НЕ. Підказка: додавати або видаляти компоненти не потрібно.
2. Створіть реалізацію схеми вентиля АБО, змінивши схему, подану на рис. 2.5. У міру необхідності можна додавати дроти, транзистори та резистори.
3. Знайдіть в Інтернеті безкоштовні програмні пакети для розробки мовою VHDL, які включають імітатор. Встановіть один із цих пакетів, налаштуйте його та спробуйте створити будь-які прості демонстраційні проєкти, що входять до складу пакета, щоб переконатися, що він працює належним чином.
4. Використовуючи набір інструментів VHDL, створіть 4-розрядний суматор за допомогою лістингів, представлених у цьому розділі.
5. Додайте у свій 4-розрядний суматор кора тестового драйвера (приклади можна знайти в Інтернеті за фразою "VHDL testbench’), потім запустіть його з обмеженим набором вхідних даних і перевірте правильність вихідних даних.
6. Розкрийте код тестового драйвера і переконайтеся, що 4-розрядний суматор дає правильні результати для всіх можливих комбінацій вхідних даних.

3

Елементи процесора

З цього розділу починається розвиток нашого всебічного розуміння архітектури сучасних процесорів. Спираючись на базові цифрові схеми, представлені в *розділі 2*, ми обговоримо функціональні блоки простого процесора для універсального комп'ютера. У цьому розділі вводяться поняття, пов'язані з набором інструкцій і набором регістрів, за якими слідує обговорення кроків, пов'язаних із завантаженням, декодуванням, виконанням і визначенням порядку проходження інструкцій. Режими адресації та категорії інструкцій обговорюються в контексті архітектури процесора 6502. Ми вирішили зосередитися на цьому процесорі, що заслужив загальне визнання, через його структурну чистоту і простоту, що дозволить нам розглянути основні концепції, не відволікаючись на другорядні аспекти. Вимоги до обробки переривань процесора вводяться на прикладі обробки переривань у процесорі 6502. Представлено стандартні підходи, які використовують сучасні процесори для операцій введення-виведення (input/output, I/O), у тому числі прямий доступ до пам'яті (direct memory access, DMA).

Прочитавши цей розділ, ви ознайомитеся з основними компонентами процесора і структурою наборів інструкцій. Ви вивчите категорії інструкцій процесора, дізнаєтеся, чому необхідна обробка переривань, і отримаєте уявлення про операції введення-виведення.

У цьому розділі буде розглянуто такі теми:

- простий процесор;
- набір інструкцій процесора;
- режими адресації;
- категорії інструкцій;
- обробка переривань;
- операції введення-виведення

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organizaйon-Second-Edition.**

Простий процесор

Архітектуру процесора 6502 і кілька його інструкцій було представлено в розділі 1. У цьому розділі ми будемо спиратися на цей фундамент, щоб представити деякі функціональні компоненти, які широко використовуються в процесорних архітектурах, – від найкрихітніших вбудованих контролерів до найпотужніших серверних процесорів.

Інтегральна схема, що лежить в основі комп'ютерної системи, має кілька різних назв: **центральний процесор** (ЦП), мікропроцесор або, просто, процесор. Мікропроцесор – це єдина інтегральна схема, що реалізує всі функції процесора. У цій книзі всі категорії центральних процесорів і мікропроцесорів будуть називатися процесорами.

- Пристрій керування здійснює загальне керування роботою процесора. Процес керування включає в себе витяг наступної інструкції з пам'яті, декодування інструкції для визначення операції, яку потрібно виконати, і розподіл дій з виконання інструкції між відповідними елементами процесора.
- Арнфметнко-логічний пристрій (АЛП) – це комбінаційна схема, яка виконує арифметичні операції та операції з бітами.
- Набір регістрів надає вихідні та кінцеві місця розташування для зберігання вхідних і вихідних даних інструкцій. Регістри також використовуються як місця для тимчасового зберігання даних.

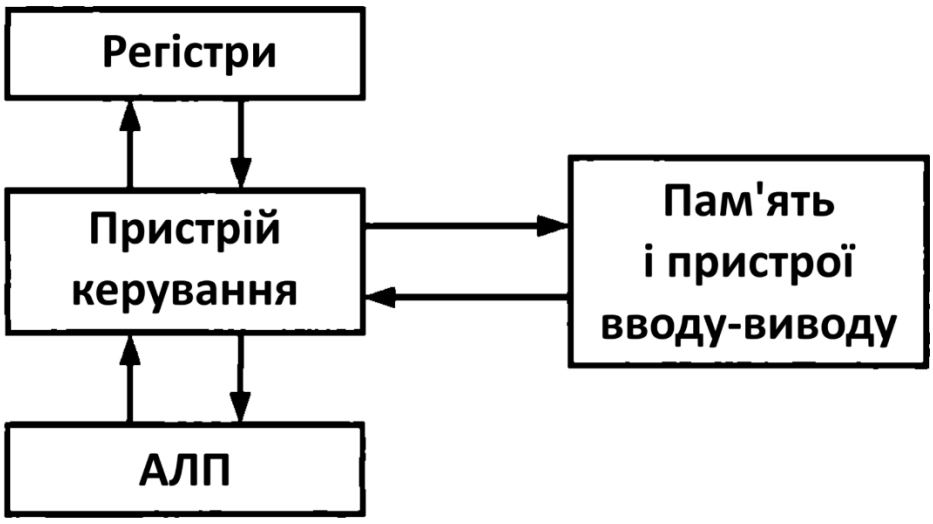


Рис 3.1. *Взаємодія між функціональними блоками процесора*

На рис. 3.1 показано потоки керування і даних між пристроєм керування, регістрами, АЛП, системною пам'яттю і пристроями введення-виведення.

Пристрій керування керує загальними операціями процесора для виконання кожної інструкції. Регістри, АЛП, пам'ять і пристрої введення-виведення реагують на команди, ініційовані пристроєм керування.

Пристрій керування

Пристрій керування сучасного процесора являє собою синхронну послідовну цифрову схему. Він інтерпретує інструкції процесора і керує виконанням цих інструкцій, взаємодіючи з іншими функціональними блоками всередині процесора і з зовнішніми компонентами, включно з пам’яттю і пристроями введення-виведення. Пристрій керування є ключовою частиною архітектури фон Неймана, реалізованої в процесорі 6502.

У цьому розділі термін "пам’ять" відноситься до **оперативного запам’ятовуючого пристрою (ОЗУ, оперативна пам’ять)**, що є зовнішнім стосовно виконавчих блоків процесора. Кеш-пам’ять, яка часто розташовується на інтегральній схемі мікропроцесора, буде розглянута в наступних розділах.

Деякі приклади пристроїв введення-виведення: комп’ютерна клавіатура, миша, дисковий накопичувач і монітор. Інші поширені пристрої введення-виведення: мережеві інтерфейси, інтерфейси бездротового зв’язку Wi-Fi і Bluetooth, аудіовихід на динаміки та мікрофонний вхід.

Після увімкнення комп’ютерної системи процесор виконує процедуру скидання для ініціалізації своїх внутрішніх компонентів. Потім процесор завантажує в **програмний лічильник** (ПЛ) адресу комірки пам’яті першої інструкції, яка має бути виконана. Розробники програмного забезпечення, які створюють системні програмні компоненти найнижчого рівня, повинні налаштувати свої засоби розробки для створення образу коду в пам’яті, виконання якого розпочинається за адресою, яку вимагає архітектура процесора.

ПЛ є центральним компонентом пристрою керування. Він завжди містить адресу інструкції, яка має бути виконана наступною. На початку кожного циклу виконання інструкції пристрій керування зчитує слово даних з комірки пам’яті за адресою, зазначеною ПЛ, і поміщає його у внутрішній регістр для декодування і виконання. Перше слово інструкції містить **код операції** (опкод). На основі бітового шаблону опкоду пристрій керування може зчитувати додаткові комірки пам’яті, що слідують за опкодом, для отримання даних, потрібних інструкцією, як-от адреса комірки пам’яті або операційний код даних.

Коли пристрій керування починає виконувати інструкції, він працює в циклі, що повторюється, показаному на рис. 3.2.

Після скидання ПЛ містить адресу комірки пам’яті з першою інструкцією. Пристрій керування витягує першу інструкцію з пам’яті та декодує її. Під час декодування пристрій керування визначає дії, які вимагає інструкція.

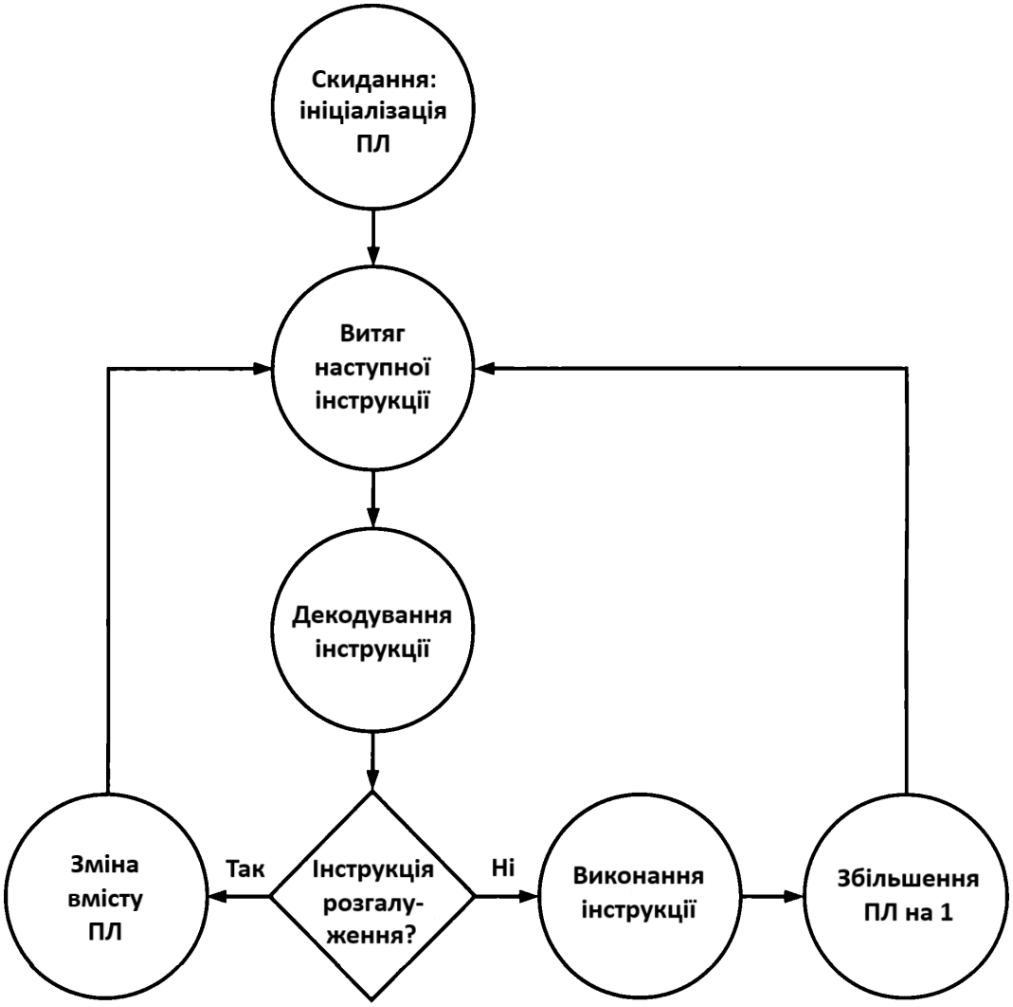


Рис 3.2. *Цикл виконання інструкції*

Також під час процесу декодування пристрій керування визначає категорію інструкції. На рис. 3.2 представлено дві основні категорії інструкцій – інструкції розгалуження та всі інші інструкції. Інструкції розгалуження реалізуються безпосередньо пристроєм керування. Вони викликають заміну вмісту ПЛ адресою комірки пам’яті для виконання гілки. Прикладом інструкцій, що виконують розгалуження, можуть бути: інструкції умовного розгалуження (коли здійснюють вибір гілки), виклики підпрограм, повернення з підпрограм та інструкції безумовного розгалуження (так званими *переходами*).

Інструкції, що не передбачають розгалуження, виконуються схемою процесора під контролем пристрою керування.

У певному сенсі для керування виконанням інструкцій без розгалуження пристрій керування використовує спосіб, аналогічний млину аналітичної машини (див. главу 1), за винятком того, що замість штифтів на барабані, що обертається, для зачеплення частин механізму млину пристрій керування використовує декодовані біти опкоду інструкції для задіяння певних частин цифрової схеми. Обрані компоненти схеми виконують завдання, які вимагає інструкція.

Процес виконання команди може містити такі дії, як читання або запис регістра, читання або запис комірки пам’яті, вказівку АЛП виконати математичну операцію та інші дії.

У більшості процесорів виконання однієї інструкції займає кілька тактових циклів. Кількість циклів для виконання інструкції може змінюватися в широких межах – від простих інструкцій, що вимагають невеликої кількості тактів, до складних операцій, виконання яких займає багато циклів. Пристрій керування координує всю цю діяльність.

Схеми, керовані пристроєм керування, побудовані з простих логічних вентилів, розглянутих у главі 2, і часто містять у собі елементи вищого рівня, як-от мультиплексори, засувки та тригери. Мультиплексори, зокрема, зазвичай використовуються логікою пристрою керування для вибіркової маршрутизації даних у задане місце призначення.

Виконання інструкції – простий приклад

Розглянемо спрощений приклад виконання двох інструкцій процесора 6502: TXA і TYA. TXA копіює вміст регістра X у регістр A, а TYA робить те саме, використовуючи як джерело регістр Y. Якщо ми розглянемо ці дві інструкції окремо від інших, то їх виконання може бути реалізовано, як показано на рис. 3.3.

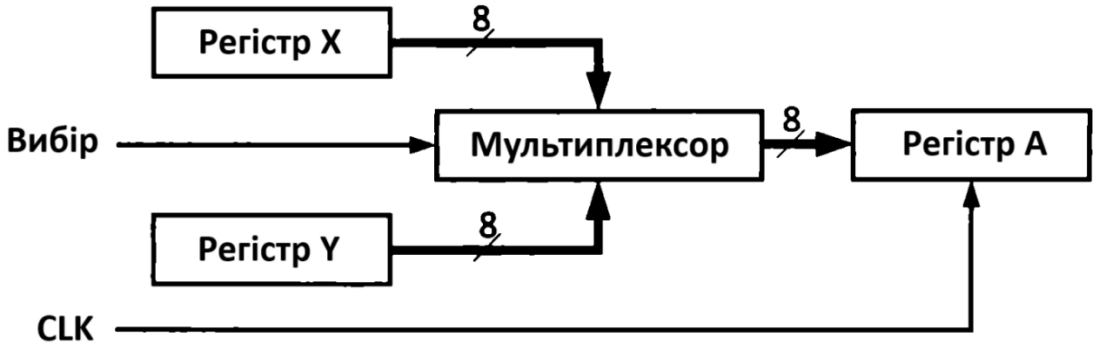


Рис 3.3. *Інструкції TXA і TYA процесора 6502*

У схемі на рис. 3.3 передбачається, що регістри X і У побудовані на основі D-тригерів (як на рис. 2.14), за винятком того, що в процесорі 6502 вони являють собою 8-бітні регістри, а не 4-бітні. Мультиплексор реалізовано у вигляді восьми копій однорозрядного мультиплексора з двома входами, як на рис. 2.9. Для керування всіма копіями використовується один загальний вхід вибору. На рис. 3.3 товстіші лінії представляють 8- розрядні шини даних, а тонші лінії – окремі логічні сигнали. Короткі штрихи, що перетинають товсті лінії з цифрою 8 над ними, вказують кількість бітів у шині.

Для виконання інструкції lm робляться такі кроки:

1. Спочатку пристрій керування подає на вхід Вибір сигнал для передання на вихід мультиплексора бітів даних регістра X. Дані з регістра X надходять на вхід регістра A.
2. Після подачі сигналу на вхід Вибір мультиплексора пристрій керування має витримати паузу, щоб біти даних досягли виходів мультиплексора.
3. Після стабілізації вихідних сигналів мультиплексора пристрій керування генерує сигнал CLK, за фронтом якого біти даних регістра X завантажуються в регістр A.

Для виконання інструкції TYA пристрій керування виконує ту саму послідовність дій, за винятком того, що спочатку він має подати на вхід **Вибір** сигнал для передачі на вихід мультиплексора даних регістра Y.

Це дуже простий приклад роботи пристрою керування з інструкціями, але він демонструє, що виконання окремої інструкції може складатися з кількох етапів і здатне задіяти лише невелику частину логічних схем, наявних у процесорі. Пристрій керування має гарантувати, що компоненти процесора, які не використовуються для виконання інструкції, перебувають у режимі простою. Це запобігає втручанню зазначених компонентів у виконання інструкції та зводить до мінімуму енергоспоживання.

Арифметико-логічний пристрій

Арифметико-логічний пристрій (АЛП) виконує арифметичні та бітові операції під керуванням пристрою керування. Для виконання операції АЛП вимагає введення значень даних, званих операндами, разом із кодом, що вказує на операцію, яка має бути виконана. Вихідні дані АЛП є результатом операції. Операції АЛП можуть використовувати як вхідні дані один або кілька прапорів процесора, таких як прапор перенесення, а також встановлювати стани прапорів процесора як вихідні дані. У процесорі 6502 операції АЛП змінюють стани прапорів перенесення, заперечення, нуля та переповнення.

АЛП – це комбінаційна схема, яка передбачає, що її виходи оновлюються асинхронно, реагуючи на зміни на входах. Крім того, цей пристрій не зберігає дані про попередні операції.

Для того щоб виконати інструкцію, що зачіпає АЛП, пристрій керування передає в АЛП вхідні дані, витримує паузу, щоб врахувати затримку проходження сигналу по колах АЛП, а потім передає вихідні дані АЛП за адресою, вказаною інструкцією.

АЛП містить схему суматора для виконання операції додавання та віднімання. У процесорі з арифметикою в додатковому коді віднімання можна реалізувати шляхом виконання заперечення в додатковому коді для правого операнда і додавання

результату до лівого операнда. З математичної точки зору, при виконанні віднімання таким чином вираз A – B перетворюється в A+ (-B).

Як ви пам'ятаєте з *глави 1*, заперечення в додатковому коді для числа зі знаком виконується шляхом інвертування всіх бітів в операнді та додавання 1 до результату. З урахуванням цієї операції віднімання, представлене у вигляді A+ (-B), перетворюється на A+ (NOT(B) + 1).

Розгляд віднімання в цій формі має прояснити використання прапора переносу в процесорі 6502 у поєднанні з інструкцією SBC. Коли запозичення під час віднімання відсутнє, прапор C забезпечує додавання 1. За наявності запозичення сума має бути зменшена на 1, що досягається шляхом скидання прапора C.

Підбиваючи підсумок, можна сказати, що в процесорі 6502 логіка віднімання ідентична логіці додавання з тією єдиною відмінністю, що операнд B у виразі A – B пропускають через набір вентилів НЕ для інвертування всіх восьми бітів перед поданням значення НЕ(B) на вхід суматора.

На рис. 3.4 показано функціональне представлення операцій додавання і віднімання в процесорі 6502.

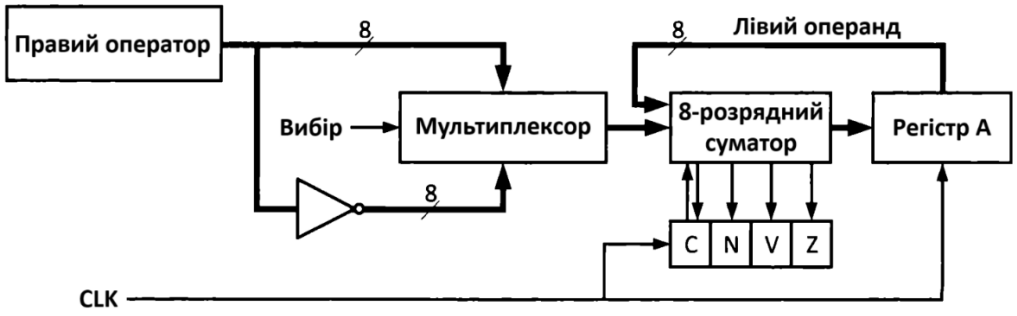


Рис 3.4. *Операції додавання та віднімання в процесорі 6502*

Як і на рис. 3.3, на рис. 3.4 показано сильно спрощене представлення процесора 6502, де є тільки ті компоненти, які беруть участь у виконанні інструкцій ADC і SBC. Вхід **Вибір** на рис. 3.4 визначає тип операції: додавання або віднімання. Для додавання потрібно вибрати верхній вхід мультиплексора, а для віднімання – нижній. В архітектурі процесора 6502 регістр A завжди містить лівий операнд для операції віднімання.

Вхідними даними для суматора є лівий і правий операнди та прапорець C. Під час виконання інструкції ADC або SBC пристрій керування передає правий операнд на входи даних мультиплексора та встановлює вхід **Вибір** мультиплексора в стан, відповідний до виконуваної інструкції. Після паузи, необхідної для проходження сигналу через вентиль НЕ, мультиплексор і суматор, пристрій керування генерує сигнал синхронізації, за фронтом якого вихідні дані суматора фіксують у регістрі A і регістрі прапорів процесора.

Прапори процесора встановлюються, як показано в наступному описі виконання інструкції ADC або SBC:

- C вказує на наявність перенесення при додаванні (C= 1) або запозичення при відніманні (C = 0);
- N містить значення біта 7 результату;
- V вказує, чи відбулося переповнення з урахуванням знака (V = 1, якщо відбулося переповнення);
- Z дорівнює 1, якщо результат дорівнює нулю; в іншому випадку Z дорівнює 0.

Крім додавання і віднімання двох чисел АЛП підтримує безліч інших операцій. У процесорі 6502 для більшості операцій з двома операндами лівий операнд поміщається в регістр A. Правий операнд або зчитується з комірки пам'яті, або надається у вигляді безпосереднього значення в комірці пам'яті, що йде за кодом операції. У процесорі 6502 усі операнди та результати АЛП є 8-бітними значеннями. Операції АЛП процесора 6502:

- ADC, SBC – додавання або віднімання двох операндів зі входом перенесення;
- DEC, DEX, DEY – зменшення значення в комірці пам'яті або регістрі на одиницю;
- INC, INX, INY – збільшення значення в комірці пам'яті або регістрі на одиницю;
- AND – побітова логічна операція І над двома операндами;
- ORA – побітова логічна операція АБО над двома операндами;
- EOR – побітова логічна операція "виключне АБО" над двома операндами;
- ASL, LSR – зсув значення в регістрі A або комірці пам'яті вліво або вправо на один розряд і вставка 0 у позицію, що звільнилася;
- ROL, ROR – обертання значення в регістрі A або комірці пам'яті ліворуч або праворуч на один розряд і вставка значення прапора C у позицію, що звільнилася;
- CMP, CPX, CPY – віднімання одного операнда від іншого і відкидання результату з встановленням прапорів N, Z і C залежно від результату віднімання;
- BIT – побітова логічна операція І над двома операндами та встановлення прапора Z, щоб вказати, чи дорівнює результат 0; крім того, копіювання бітів 7 і 6 лівого операнда у прапори N і V.

Можливості АЛП процесора 6502 обмежені порівняно зі складнішими сучасними процесорами, такими як сімейство x86.

Наприклад, у 6502 операції множення і ділення програміст повинен реалізувати в коді у вигляді повторюваних операцій додавання і віднімання. Крім того, процесор 6502 може виконувати зсув або обертання значення в рамках однієї інструкції всього на один розряд.

У процесорах сімейства x86, з іншого боку, інструкції множення і ділення реалізовано без посередників, а інструкції зсуву і обертання включають параметр, що вказує кількість розрядів для зсуву в рамах однієї інструкції.

АЛП – це неминуче складний логічний пристрій, що робить його ідеальним кандидатом для проектування за допомогою мови опису апаратних засобів. Наступний лістинг – це реалізація мовою VHDL частини АЛП, подібного до того, що застосовується в процесорі 6502:

```
-- Завантаження стандартних бібліотек
library IEEE;
use IEEE.STD_L0GIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- Визначення входів і виходів 8-розрядного АЛП
entity ALP is
 port (
   -- Лівий операнд
   LEFT : in std_logic_vector(7 downto 0);
   -- Правий операнд
   RIGHT : in std_logic_vector(7 downto 0);
   -- Операція АЛП
   OPCODE : in std_logic_vector(3 downto 0);
   -- Перенесення з попереднього розряду
   C_IN : in std_logic;
   -- Результат АЛП
   RESULT : out std_logic_vector(7 downto 0);
   -- Перенос в следующий разряд
   C_OUT : out std_logic;
   -- Виведення прапора знака
   N_OUT out std_logic;
   -- Виведення прапора переповнення
   V_OUT : out std_logic;
   -- Виведення прапора нуля
   Z_ OUT out std_logic
 );
end entity ALP;

-- Визначення поведінки 8-розрядного АЛП
```

```
architecture BEHAVIORAL of ALP is

begin

 P_ALP : process (LEFT, RIGHT, OPCODE, C_IN) is

  variable result8 : unsigned(7 downto 0);
  variable result9 : unsigned(8 downto 0);
  variable right_op : unsigned(7 downto 0);

begin

  case OPCODE is

    when "0000" | "0001" =>             -- Додавання або віднімання

      if (OPCODE = "0000") then
       right_op := unsigned(RIGHT);     -- Додавання
      else
       right_op := unsigned(not RIGHT); -- Віднімання
      end if;

      result9 := ('0' & unsigned(LEFT)) +
                 unsigned(right_op) +
                 unsigned(std_logic_vector'(""& C_IN));
      result8 := result9(7 downto 0);

      C_OUT <= result9(8); -- Прапор C

      -- Прапор V
      if (((LEFT(7) XOR result8(7)) = '1') AND

         ((right_op(7) XOR result8(7)) = '1')) then
      V_OUT <= '1';
     else
      V_OUT <= '0';
     end if;
```

```
      when "0010" => -- Збільшення на 1
       result8 := unsigned(LEFT) + 1;
      when "0011" => -- Зменшення на 1
       result8 := unsigned(LEFT) - 1;
      when "0101" => -- Побитова операція I
       result8 := unsigned(LEFT and RIGHT);
      when "0110" => -- Побитова операція АБО
       result8 := unsigned(LEFT or RIGHT);
      when "0111" => -- Побитова операція виключна АБО
       result8 := unsigned(LEFT xor RIGHT);
      when others =>
       result8 := (others => 'X');
    end case;

    RESULT <= std_logic_vector(result8);

    N_OUT <= result8(7); =>              -- Прапор N

    if (result8 = 0) then                -- Прапор Z
     Z_OUT <= '1';
    else
     Z_OUT <= '0';
    end if;

   end process P_ALP;

  end architecture BEHAVIORAL;
```

Цей код визначає просте АЛУ як комбінаційну схему з лівим операндом, правим операндом, кодом операції та прапором C як вхідні дані. Вихідними даними є результат операції та прапори C, N, V і Z.

Далі ми розглянемо призначення та функції регістрів процесора.

Регістри

Регістри процесора – це внутрішні області зберігання даних, які виконують функції джерел і місць призначення для операцій, що використовуються для виконання інструкцій. Регістри забезпечують найшвидший доступ до даних у процесорі, але

обмежені дуже невеликою кількістю комірок пам’яті через їхню високу вартість з погляду площі матриці. Розрядність регістра зазвичай збігається з розміром слова процесора.

Процесор 6502, як ми вже бачили, має тільки три 8-розрядних регістри: A, X, і Y. Процесори сімейства x86 мають шість 32-розрядних регістрів для тимчасового зберігання даних: EAX, EBX, ECX, EDX, ESI та EDI. В архітектурах багатьох процесорів певні регістри виділяються для передавання вхідних даних, необхідних певними інструкціями. Наприклад, в архітектурі x86 одна інструкція REP MOVSD переміщує блок даних, довжина якого (у словах) зберігається в регістрі ECX, що починається з адреси джерела з регістра ESI, за адресою призначення з регістра EDI.

Під час розроблення нової архітектури процесора вкрай важливо знайти золоту середину між кількістю регістрів і кількістю та складністю інструкцій, доступних процесору. Якщо взяти кристал інтегральної схеми певного розміру і певної технології виготовлення (обидва ці фактори обмежують кількість транзисторів, доступних для процесора), то додавання більшої кількості регістрів в архітектуру зменшує кількість транзисторів, доступних для виконання інструкцій та інших функцій. З іншого боку, додавання інструкцій зі складними можливостями здатне обмежити простір кристала, доступний для регістрів. Це протиріччя між складністю набору інструкцій і кількістю регістрів виражається у віднесенні архітектури до категорії CISC або RISC.

- Процесори категорії **CISC (complex instrucycyon set copruter – комп’ютер з повним набором інструкцій)** характеризуються наявністю великого набору інструкцій, що підтримує безліч функцій, таких як можливість завантажувати операнди з пам’яті, виконувати операцію і зберігати результат у пам’яті, і все це за допомогою однієї інструкції. У процесорі CISC виконання інструкції може зайняти багато тактів, поки процесор виконує всі необхідні підзадачі. Згадана вище інструкція REPMOVSD є прикладом інструкції з потенційно тривалим часом виконання. Процесори CISC, як правило, мають меншу кількість регістрів, почасти через те, що значну площу кристала займають схемні елементи, що реалізують логіку набору інструкцій. Сімейство процесорів x86 є класичним прикладом архітектури CISC.
- Процесори категорії **RISC (reduced instruction set computer – комп’ютер з скороченим набором інструкцій)**, навпаки, мають меншу кількість інструкцій, кожна з яких виконує простіші завдання порівняно з інструкціями процесорів CISC. Виконання операції зі значеннями даних, що зберігаються в пам’яті, може зажадати пари інструкцій для завантаження двох операндів з пам’яті в регістри, ще однієї інструкції для виконання операції та заключної інструкції для запису результату операції в пам’ять.

Ключова відмінність між CISC і RISC полягає в тому, що архітектури RISC оптимізовані для виконання окремих інструкцій з дуже високою швидкістю.

Незважаючи на те, що в RISC процесорі для читання даних з пам'яті, виконання операції та запису результату в пам'ять потрібно на кілька інструкцій більше, ніж у CISC-процесорі, загальний час від початку до кінця виконання тих самих операцій може бути порівнянним або навіть меншим у разі RISC процесора. Прикладами категорії RISC можуть слугувати архітектури ARM, які обговорюють у *розділі 10*, або архітектура RISC-V, яку розглядають у *розділі 11*.

Спрощення набору інструкцій у RISC-процесорах залишає більше місця на кристалі для регістрів, це означає, що в RISC-процесорах зазвичай використовується більша кількість регістрів порівняно з CISC-процесорами. Архітектура ARM, наприклад, має 13 регістрів загального призначення, тоді як базова 32-розрядна архітектура RISC-V – 31 регістр загального призначення.

Більша кількість регістрів в архітектурах RISC зменшує потребу в доступі до системної пам'яті, оскільки для зберігання проміжних результатів доступно більше регістрів. Це підвищує продуктивність системи, оскільки доступ до системної пам'яті займає значно більше часу, ніж доступ до даних, розташованих у регіетрах процесора.

Уявіть собі регістр процесора як набір D-тригерів, у яких кожен тригер містить один біт даних регістра. Дані завантажуються в кожен із тригерів регістра за загальним тактовим сигналом. Дані для запису в регістр можуть надходити в тригери після проходження через мультиплексор, який обирає одне з потенційно багатьох джерел даних, дотримуючись вказівок інструкції, що виконується.

Як альтернатива використанню мультиплексора для цієї мети інструкція може завантажувати дані в регістр із шини передавання даних, внутрішньої по відношенню до процесора. У цій конфігурації пристрій керування організовує роботу внутрішньої шини таким чином, щоб тільки потрібне джерело даних керувало лініями шини в момент проходження фронту тактового сигналу, який завантажує дані до регістра, тоді як усім іншим джерелам даних, під'єднаним до шини, заборонено розміщувати на ній дані.

У наступних розділах буде представлено повний склад набору інструкцій процесора та використовувані інструкціями режими адресації.

Набір інструкцій процесора

Інструкції, аналогічні тим, що обговорювалися вище, реалізовано в більшості архітектур процесорів загального призначення, однак можливості складніших процесорів розширюють за рахунок додаткових категорій інструкцій. Більш досконалі інструкції, доступні в архітектурах сучасних процесорів, будуть представлені в наступних розділах.

Процесори CISC зазвичай підтримують кілька режимів адресації. Вони, своєю чергою, призначені для ефективного доступу до послідовних комірок пам'яті, що

використовуються програмними алгоритмами, які працюють на процесорі. У наступному розділі описуються режими адресації інструкцій, реалізовані в процесорі 6502. Далі будуть представлені категорії інструкцій процесора 6502, більшість з яких застосовуються в архітектурах сучасних процесорів. Потім ми розглянемо спеціалізовані інструкції для обробки переривань та операцій вводу-виводу, включно з поясненням функцій процесора, що забезпечують високопродуктивні операції вводу та виводу з блоками даних змінного розміру.

Режими адресації

CISC-процесори підтримують кілька режимів адресації для інструкцій, які потребують передавання даних між пам'яттю та регістрами. RISC-процесори мають обмежену кількість режимів адресації. Кожна архітектура визначає свій набір режимів адресації на основі аналізу передбачуваних моделей доступу до пам'яті, які програмне забезпечення буде використовувати в цій архітектурі.

Для того щоб уявити режими адресації процесора 6502, у цьому розділі використовується простий приклад кода 6502, що складає чотири байти даних. Для того щоб уникнути зайвих деталей, у прикладі ігнорується будь-яке перенесення з 8-бітної суми.

Режим безпосередньої адресації

У разі неопосередкованої адресації значення операції слідує в пам'яті відразу за кодом операції. Для першого прикладу припустімо, що нам дано значення чотирьох байтів для додавання і запропоновано написати для процесора 6502 програму, що розв'язує цю задачу. Така постановка задачі дозволила нам вводити байтові значення безпосередньо в наш код. Байти в цьому прикладі – від $01 до $04. Ми складатимемо байти у зворотному порядку (від $04 до $01), передбачаючи далекий перехід до використання циклічної конструкції, що буде представлена в розділі нижче. У цьому коді для додавання чотирьох байтів використовується **режим опосередкованої адресації:**

```
; Додавання чотирьох байтів з використанням режиму безпосередньої
адресації
LDA #$04
CLC
ADC #$03
ADC #$02
ADC #$01
```

Зверніть увагу, що коментарі мовою асемблера починаються з крапки з комою. Після виконання цих інструкцій регістр A міститиме значення $0A – суму чотирьох байтів, зазначених як операнди.

Згадайте *главу 1* – там було сказано, що в мові асемблера 6502 безпосередньому значенню передує символ #, а символ $ вказує, що значення є шістнадцятковим. Безпосередньо адресований операнд зчитується з комірки пам'яті за адресою, яка йде за кодом операції інструкції. Безпосередня адресація зручна тим, що при її використанні немає необхідності резервувати комірку пам'яті, з якої слід зчитувати операнд. Однак режим безпосередньої адресації корисний тільки в тому разі, якщо значення даних відоме під час написання програми.

Режим абсолютної адресації

У **режимі абсолютної адресації**, іноді званому **режимом прямої адресації**, вказується осередок пам'яті, що містить значення, яке буде зчитано або записано інструкцією. У процесорі 6502 передбачено 16 адресних бітів, тому адресне поле, що підтримує доступ до всієї доступної пам'яті, має довжину 2 байти. Повна інструкція для доступу до даних у довільному осередку пам'яті процесора 6502 складається з трьох байтів: перший байт – це код операції, за ним ідуть два байти з адресою осередку пам'яті для зчитування або запису. При збереженні двох байтів адреси першим розташовується молодший байт, а за ним слідує старший байт. Старший байт 16-розрядного значення містить 8 старших значущих бітів, а молодший байт – 8 молодших значущих бітів.

Завдяки прийнятій угоді про зберігання молодшого байта двобайтової адреси за меншою адресою пам'яті 6502 є процесором **із прямим порядком байтів** (little-endian). У процесорах сімейства x86 також прийнято прямий порядок байтів. Архітектури ARM і RISC-V дають змогу програмно обирати режим прямого або зворотного порядку проходження байтів (**порядок байтів, що перемикається** (bi-endianness)), але більшість операційних систем, що працюють на цих архітектурах, обирають режим із прямим порядком байтів.

Приклад режиму абсолютної адресації ми почнемо з деякого підготовчого коду для збереження чотирьох байтів, які будуть складатися, за адресами від $200 до $203. За підготовчим кодом слідують інструкції для складання цих чотирьох байтів. У цьому прикладі для складання чотирьох байтів застосовується режим абсолютної адресації:

```
; Ініціалізація даних у пам'яті
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
```

```
STA $0200
; Додавання чотирьох байтів з використанням режиму абсолютної адресації
LDA $0203
CLC
ADC $0202
ADC $0201
ADC $0200
```

На відміну від режиму безпосередньої адресації, абсолютна адресація дозволяє підсумувати чотири значення, які не відомі до виконання програми:

інструкції ADC складають будь-які значення, збережені в комірках $200-$203. Обмеження цього режиму адресації полягає в тому, що під час написання програми необхідно вказати адреси байтів, що складаються. Той код не може складати байти, розташовані в довільному місці в пам’яті.

Недоліком цього простого прикладу є використання надлишкової послідовності майже ідентичних інструкцій. Для того щоб цього уникнути, зазвичай бажано помістити повторювану частину коду в циклічну конструкцію. У наступних двох прикладах використовується режим адресації 6502, який полегшує виконання циклічних операцій, хоча ми не будемо додавати в код цикл до другого прикладу.

Режим непрямої індексної адресації

У **режимі абсолютної індексної адресації** адресу комірки пам’яті обчислюють шляхом додавання до базової адреси, зазначеної в інструкції, значення, яке міститься в регістрі X або У. У наступному прикладі за допомогою абсолютної індексної адресації складаються байти за адресами від $0200 до $0203. Регістр X надає зміщення від початку масиву байтів за адресою $0200:

```
; Ініціалізація даних у пам’яті
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
```

```
; Додавання чотирьох байтів з використанням режиму абсолютної індексної адресації
LDX #$03
CLC
LDA $0200, X
DEX
ADC $0200, X
DEX
ADC $0200, X
DEX
ADC $0200, X
```

Інструкція DEX зменшує значення в регістрі X на одиницю (віднімає від нього 1). Цей код погіршує рішення з точки зору збільшення кількості інструкцій, необхідних для додавання чотирьох байтів, однак ми бачимо, що послідовність з інструкції DEX і наступної за нею інструкції ADC $0200, x тепер повторюється тричі.

Ми можемо використовувати умовне розгалуження для виконання цієї послідовності додавання в циклі:

```
; Ініціалізація даних у пам’яті
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Додавання чотирьох байтів з використанням режиму абсолютної індексної адресації
LDX #$03
LDA $0200, X
DEX
CLC
ADD_LOOP:
ADC $0200, X
```

```
DEX
BPL ADD_LOOP
```

Інструкція BPL (розгалуження при плюсі) здійснює умовну передачу інструкції, перед якою стоїть мітка ADD_LOOP. BPL виконує це розгалуження тільки тоді, коли прапор N процесора дорівнює нулю. Якщо прапор N встановлено, інструкція BPL передає керування наступній інструкції в пам’яті.

Може здатися, що код у цьому прикладі не вартий зусиль, витрачених на створення циклу лише для того, щоб скласти чотири байти. Але зверніть увагу, що цю версію коду можна змінити для додавання 100 послідовних байтів простою зміною операнда інструкції LDX. Зміна попереднього прикладу для додавання 100 байтів у такий самий спосіб зажадала б набагато більше роботи, і інструкції споживали б набагато більше пам’яті.

Цей приклад має ті самі обмеження, що й приклад для режиму абсолютної адреси – в обох випадках задано початок масиву байтів у комірці пам’яті, визначеній під час написання програми. Наступний режим адресації усуває це обмеження і складає масив байтів, починаючи з будь-якої адреси в пам’яті.

Режими непрямої індексної адреси

Режим непрямої індексної адресації використовує двобайтову адресу, що зберігається в діапазоні адрес пам’яті $00-$FF, як базову адресу і додає вміст регістра У до цієї бази для одержання адреси пам’яті, використовуваної інструкцією. У наступному прикладі базова адреса масиву байтів ($Ф2вв) попередньо зберігається за адресами $0010 і $0011 з дотриманням прямого порядку байтів. Наступний код використовує для додавання байтів непряму індексну адресацію в циклі:

```
; Ініціалізація даних у пам’яті
LDA #$04
STA $0203
LDA #$03
STA $0202
LDA #$02
STA $0201
LDA #$01
STA $0200
; Ініціалізація покажчика на масив байтів
LDA #$00
STA $10
```

```
LDA #$02
STA $11
; Додавання чотирьох байтів з використанням режиму непрямої індексної адресації
LDY #$03
LDA ($10), Y
DEY
CLC
ADD_LOOP:
ADC ($10), Y
DEY
BPL ADD_LOOP
```

У разі непрямої індексної адресації перед виконанням коду додавання можна зберегти будь-яку адресу в пам’яті за адресами $10-$11. Зверніть увагу, що під час непрямої індексної адресації для зберігання зсуву адреси необхідно використовувати регістр У. Регістр Х недоступний для використання в цьому режимі адресації.

У процесорі 6502 передбачені й деякі інші режими адресації: **режим адресації за нульовою сторінкою** (абсолютною та абсолютною з індексуванням) дає змогу використовувати інструкції, які мають меншу довжину (на один байт) і виконуються швидше за рахунок роботи тільки з адресами пам’яті в діапазоні $00-$FF. Термін *«нульова сторінка»* пов’язаний із тим, що для адрес у цьому діапазоні старший байт 16-розрядної адреси дорівнює нулю. Крім поліпшення продуктивності завдяки вищій швидкості виконання і зменшення пам’яті, яку займає код, режими адресації за нульовою сторінкою діють так само, як і відповідні режими адресації, описані раніше.

Інший режим називається **режимом індексної непрямої адресації** [1], він схожий на режим непрямої індексної адресації, за винятком того, що замість регістра У використовується регістр Х, а зсув, який міститься в регістрі Х, додається до адреси, зазначеної в інструкції, для визначення адреси покажчика на дані. Наприклад, припустимо, що регістр Х містить значення 8. Інструкція LDA ($10, X) додає $10 до вмісту регістра Х, що дає в результаті $18. Потім ця інструкція використовує 16-розрядну адресу пам’яті, зчитану з пам’яті за адресами $18-$19, як адресу цільового осередку пам’яті для завантаження значення в регістр А.

[1] Режими індексної непрямої адресації та непрямої індексної адресації названо за аналогією до англійських назв, обраних автором ("indexed indirect addressing" та "indirect indexed addressing"), їхні назви не відображають відмінностей між ними. – Пер.

Індексна непряма адресація не має відношення до нашого прикладу підсумовування послідовності байтів. Одним із прикладів застосування цього режиму є вибір значення з послідовного списку покажчиків, де кожен покажчик містить адресу рядка символів. Регістр X може посилатися на один із цих рядків як на зміщення від початку списку покажчиків. Така інструкція, як LDA ($10, X), завантажує в регістр A адресу обраного рядка.

Режиги адресації, доступні в архітектурах процесорів CISC і, меншою мірою, в архітектурах RISC, забезпечують підтримку ефективних методів доступу до різних типів структур даних у системній пам’яті. У наступному розділі обговорюються категорії інструкцій, реалізовані в архітектурі процесора 6502, а також використання кожною інструкцією доступних режимів адресації.

Категорії інструкцій

У цьому розділі представлено категорії інструкцій, реалізовані в процесорі 6502. Мета обговорення процесора 6502 тут полягає в тому, щоб представити концепції, пов’язані з набором інструкцій більш простої процесорної архітектори, ніж у сучасних 32- і 64-розрядних процесорів, які ми розглянемо в наступних розділах. На той час, коли ми доберемося до цих процесорів, базові концепції набору інструкцій мають бути вам добре знайомі.

Інструкції завантаження та збереження

Процесор 6502 використовує інструкції завантаження з пам’яті та збереження в пам’ять для зчитування значень даних із системної пам’яті в регістри процесора та для запису вмісту регістрів у системну пам’ять. В архітектурі 6502 інструкції LDA, LDX і LDY завантажують у регістр, зазначений у їхній назві, 8-розрядне слово із системної пам’яті. Інструкція LDA підтримує всі режими адресації, доступні в 6502, а LDX і LDY – лише частину режимів адресації: безпосередню, абсолютну й абсолютну індексну.

Після завершення виконання кожної з цих інструкцій прапори N і Z вказують, чи є завантажене значення від’ємним (за встановленим бітом 7) і чи дорівнює це значення нулю.

Інструкції STA, STX і STY записують у пам’ять вміст регістра, зазначеного в їхніх назвах. Кожна інструкція збереження підтримує ті самі режими адресації, що й інструкція завантаження для цього регістра, за винятком режиму нескладної адресації. Ці інструкції оновлюють стан прапорів N і Z, що відображають властивості збереженого значення.

Інструкції передавання даних із регістра в регістр

Ці інструкції копіюють 8-бітове слово одного регістра (A, X або У) у інший регістр. Дані інструкції використовують **режим неявної адресації**, тобто джерело й адресат кожної інструкції вказуються безпосередньо кодом операції інструкції.

Інструкція ти копіює вміст регістра A в регістр X. Інструкції TAY, TXE і TYA виконують аналогічні операції між парами регістрів, зазначеними в їхніх назвах. Ці інструкції оновлюють стан прапорів N і Z.

Інструкції стека

Інструкція TXS копіює вміст регістра X у регістр покажчика стека (S). Вона має використовуватися для ініціалізації регістра S під час запуску системи. Інструкція TSX копіює вміст регістра S у регістр X і оновлює стан прапорів N і Z. Виконання інструкції TXS впливає на будь-які інші прапори.

Інструкція PHA поміщає в стек вміст регістра A. Інструкція PHP поміщає в стек прапори процесора у вигляді 8-бітного слова. Ці інструкції не впливають на прапори процесора. Поміщення значення в стек складається із запису вмісту регістра в комірку пам’яті за адресою, обчисленою шляхом додавання $l00 до вмісту регістра S, і подальшого зменшення значення регістра S на одиницю.

Інструкції PTA і PLP витягують значення зі стека і поміщають його в регістр A або регістр прапорів, відповідно. При вилученні значення до вмісту регістра S спочатку додається одиниця, після чого в цільовий регістр передається значення з комірки пам’яті за адресою, обчисленою шляхом додавання $100 до вмісту регістра S. Інструкція PLA оновлює стан прапорів N і Z. Інструкція PLP встановлює або скидає шість із семи прапорів процесора залежно від значення, витягнутого зі стека. Прапор B (прапор програмного переривання) має сенс тільки в копії регістра прапорів процесора, яка була поміщена в стек перериванням або інструкцією PHP. Це відрізняє інструкцію BRK від запиту апаратного переривання. Обидві інструкції, PHP і BRK, передають регістр прапорів зі встановленим бітом B (біт 4).

Апаратні переривання, генеровані процесором на виводах IRQ (Intemipt Request – запит на переривання) і NMI (Non-Maskable Intemipt – немасковане переривання), передають регістр прапорів процесора зі скинутим бітом B. Оброблення переривань та інструкцію BRK буде розглянуто пізніше в цьому розділі.

Арифметичні інструкції

Як ми вже бачили, додавання і віднімання виконуються за допомогою інструкцій ADC і SBC. Лівим операндом кожної інструкції виступає вміст регістра A, який також є місцем призначення для результату операції. Для зазначення правого операнда

доступні всі режими адресації. Значення прапорів Z, C, N і V оновлюються, відображаючи результат операції.

Інструкції INC і DEC, відповідно, збільшують або зменшують значення у вказаній комірці пам'яті на одиницю. Результат зберігається в тій самій комірці пам'яті. Підтримуються режими абсолютної адресації та абсолютної індексної адресації. Ці інструкції оновлюють стан прапорів N і Z залежно від результату операції.

Інструкції INX, DEX, INY і DEY збільшують або зменшують значення в регістрі X або Y на одиницю, відповідно до їхніх назв. Ці інструкції оновлюють стан прапорів N і Z.

Інструкція CMP виконує порівняння шляхом віднімання значення операнда від значення в регістрі A. Поведінка інструкції CMP дуже схожа на послідовність, у якій за інструкцією SEC слідує інструкція SBC. Значення прапорів N, Z і C оновлюється, відображаючи результат віднімання. Відмінності між інструкціями CMP і SBC полягають у такому:

- CMP відкидає результат віднімання (виконання інструкції CMP не впливає на значення в регістрі A);
- CMP не використовує десятковий режим, якщо встановлено прапор D;
- CMP не впливає на стан прапора V;
- CMP підтримує всі режими адресації.

Інструкції CPX і CPY схожі на CMP, за винятком того, що вміст регістра X або Y використовується як лівий операнд, згідно з назвою інструкції, а для правого операнда підтримуються тільки режими абсолютної адресації та абсолютної індексної адресації.

Логічні інструкції

Інструкції AND, EOR і ORA виконують побітові операції I, «виключне АБО» і АБО, відповідно, між регістром A і операндом. Результат зберігається в регістрі A. Значення прапорів Z і N оновлюються, відображаючи результат операції. Підтримуються всі режими адресації.

Інструкція ASL виконує зсув операнда на один біт вліво, вставляючи 0 у молодший біт. Старший біт зсувається у прапор C. Це еквівалент множення значення в регістрі A на 2 і перенесення старшого біта 9-бітного результату в C.

Аналогічно ASL інструкція LSR виконує зсув операнда на один біт вправо, вставляючи 0 у старший біт. Молодший біт зсувається у прапор C. Це еквівалентно діленню беззнакового операнда на 2 зі зсувом залишку в C.

Інструкції ROL і ROR зсувають вміст регістра A на один біт вліво або вправо відповідно. Попереднє значення прапора C переноситься в позицію розряду, що звільнився в результаті операції зсуву. Біт, що випадає в результаті зсуву з регістра A, зберігається у прапорі C.

Інструкції ASL, LSR, ROL і ROR підтримують режим адресації акумулятора, у якому як операнд використовується регістр A. Цей режим задається за допомогою спеціального операнда зі значенням «A», як в інструкції ASL A. Ці чотири інструкції також підтримують режими абсолютної адресації та абсолютної індексної адресації.

Інструкція BIT виконує побітову операцію I між операндом і регістром A, результат при цьому відкидається. Стан прапора Z оновлюється залежно від результату операції. Біти 7 і 6 з комірки пам'яті копіюються у прапори N і V відповідно. Підтримується тільки режим абсолютної адресації.

Інструкції розгалуження

Інструкція IMP завантажує операнд у ПС, і виконання програми триває з інструкції в зазначеному місці. Адреса призначення, двобайтова абсолютна адреса, може знаходитися в будь-якому місці адресного простору процесора 6502:

- інструкції вага і BCS виконують умовне розгалуження, якщо прапор C скинутий або встановлений, відповідно;
- інструкції BNE і BEQ виконують умовне розгалуження, якщо прапор Z скинутий або встановлений, відповідно;
- інструкції BPL і вмі виконують умовне розгалуження, якщо прапор N скинутий або встановлений, відповідно;
- інструкції BVC I BVS виконують умовне розгалуження, якщо прапор V скинутий або встановлений, відповідно.

Інструкції умовного розгалуження використовують режим відносної адресації, де цільова адреса – це 8-бітове зміщення зі знаком (у діапазоні від -128 до +127 байт) від адреси інструкції, що йде за інструкцією розгалуження.

Інструкції виклику підпрограми та повернення з підпрограми

Інструкція JSR передає в стек адресу інструкції, наступної за інструкцією JSR (мінус один), завантажує адресу, зазначену як 16-розрядний операнд, у ПС, а потім продовжує виконання програми, починаючи з інструкції в цьому місці розташування.

Інструкція RTS використовується для завершення підпрограми. Значення ПС (мінус один), що повертається, витягується зі стека і завантажується ПС. Дана інструкція збільшує значення НС на одиницю до того, як воно буде в якості адреси наступної команди для виконання.

Інструкції для роботи з прапорами процесора

Інструкції для роботи з прапорами процесора використовуються для прямого встановлення або скидання окремих прапорів.

- Інструкції SEC і CLC встановлюють і скидають прапор C, відповідно.
- Інструкції SED і CLD встановлюють і скидають прапор D, відповідно.
- Інструкція CLV скидає прапор V.

Жодна з інструкцій не встановлює прапор V.

Інструкції для роботи з перериваннями

Інструкції для роботи з перериваннями дають змогу процесору керувати обробкою зовнішніх переривань і генерувати власні програмні переривання. Зовнішні переривання можуть бути двох типів: масковані та немасковані. Переривання кожного типу запускаються за допомогою виділеного для цієї мети вхідного контакту на процесорі 6502.

Масковані переривання можна відключити, встановивши прапор I процесора. Під час маскування процесор ігнорує сигнал на відповідному вхідному контакті. Немасковані переривання, як випливає з їхньої назви, не можуть бути заблоковані і завжди ініціюють переривання процесора, коли на відповідному контакті виявляється відповідний перехід сигналу. Обробку переривань ми розглянемо детальніше в наступному розділі.

Інструкції SEI і CLI встановлюють і скидають прапор I, відповідно. Коли прапор I встановлено, масковані переривання вимкнено або масковано.

Інструкція BRK генерує немасковане переривання. Адреса пам’яті через два байти після інструкції BRK поміщається в стек, за нею йде регістр прапорів процесора. У ПС завантажується адреса обробника переривань, яка зчитується за адресами $FFFE-$FFFF. Потім запускається виконання обробника переривань.

Інструкція BRK не змінює вміст регістрів (крім покажчика стека) або прапорів процесора. Біт B регістра прапорів, поміщеного в стек, встановлено, вказуючи на те, що переривання є результатом виконання інструкції BRK.

Інструкція RTI виконує повернення з обробника переривань. Ця інструкція відновлює прапори процесора зі стека, а також відновлює стан ПС. Після відновлення прапорів процесора прапор B втрачає своє значення, і його слід ігнорувати.

Обробка переривань і використання інструкції BRK будуть обговорюватися в розд. *«Обробка переривань» далі в цьому розділі*.

Інструкція відсутності операцій

Інструкція NOP (яку також часто називають no-op) нічого не робить, крім переведення ПС до наступної інструкції.

Інструкції NOP іноді використовуються при розробці програм як засіб налагодження. Наприклад, одну або кілька інструкцій можна ефективно «закоментувати», заповнивши адреси пам'яті для цих інструкцій байтами $EA. $EA – це шістнадцяткове значення коду операції NOP у процесорі 6502.

Обробка переривань

Процесори зазвичай підтримують деяку форму обробки преривань для реагування на запити обслуговування від зовнішніх пристроїв. Концептуально обробка переривань нагадує сценарій, у якому ви зайняті розв'язанням певного завдання, і ваш телефон дзвонить. Але після відповіді на виклик і, можливо, після завдання наступної дії (наприклад, "купити хліб") ви повертаєтеся до перерваної роботи над завданням. Люди використовують кілька подібних механізмів, як-от дверні дзвінки та будильники, що дають нам змогу переривати менш пріоритетні дії та реагувати на більш нагальні потреби.

Обробка $\overline{IRQ}$

Інтегральна схема 6502 має два входи для сигналів, за допомогою яких зовнішні компоненти повідомляють процесор про необхідність приділити увагу певним подіям. Перший – це вхід запиту на переривання, $\overline{IRQ}$ (Interrupt ReQuest). Вхід $\overline{IRQ}$ є інверсним, тобто викликає переривання процесора за низького рівня (це означає, що сигнал вважається активним за низького рівня або в стані 0, на що вказує лінія над символами IRQ). Можна уявити цей сигнал як телефонний дзвінок, що повідомляє процесор про вхідний виклик.

Процесор 6502 не може миттєво реагувати на низький рівень сигналу на вході $\overline{IRQ}$. Перш ніж процесор 6502 зможе приступити до обробки переривання, він повинен завершити роботу над інструкцією, яка вже виконується. Потім він поміщає в стек адресу повернення (адреса наступної інструкції, яка мала бути виконана після поточної інструкції), за якою слідує регістр прапорів процесора. Оскільки це переривання було згенероване за допомогою входу $\overline{IRQ}$, прапор B у рядку прапорів процесора, вміщеному в стек, дорівнюватиме 0.

На відміну від інструкції JSR, адреса повернення, поміщена в стек у відповідь на сигнал зі входу $\overline{IRQ}$, є фактичною адресою наступної інструкції, яку має бути виконано, а не адресою інструкції за вирахуванням одиниці. Адреса повернення після переривання не буде збільшуватися на одиницю, як це відбувається під час формування адреси повернення під час виконання інструкції RTS.

На наступному етапі опрацювання переривання процесор завантажує в ПС адресу програми опрацювання $\overline{IRQ}$, яка зчитується з пам'яті за адресами $FFFE–$FFFF. Потім процесор 6502 починає виконання коду обробника переривання за цією адресою. Оброблювач переривання – це код, який ідентифікує периферійний пристрій, що ініціював переривання, і виконує обробку, необхідну для задоволення запиту, а потім повертає керування коду, який виконувався до переривання.

Але після завершення роботи обробника переривань процесор виконує інструкцію RTI. Вона витягує прапори процесора і стан ПС зі стека, після чого процесор відновлює виконання програми, починаючи з інструкції, що слідувала за поточною інструкцією під час переходу сигналу на вході $\overline{IRQ}$ на низький рівень.

Вхід $\overline{IRQ}$ генерує **масковане переривання**, яке означає, що в цьому випадку можна виконати дію, еквівалентну відключенню звуку під час надходження телефонного виклику. Коли починається обробка переривання $\overline{IRQ}$, процесор 6502 автоматично встановлює прапор I, який маскує (відключає) вхід $\overline{IRQ}$ до зняття цього прапора.

Прапор I скидається, коли інструкція RTI відновлює прапори процесора, оскільки цей прапор не міг бути встановлений у момент реагування процесора на сигнал $\overline{IRQ}$. Прапор I також можна зняти за допомогою інструкції CLI, це означає, що під час обробки одного переривання $\overline{IRQ}$ можна дозволити реагування на інші переривання $\overline{IRQ}$. Переривання, що обробляється під час обробки іншого переривання, називається вкладеним перериванням.

Вхід $\overline{IRQ}$ це пристрій, що запускається за рівнем сигналу. Це означає, що щоразу, коли сигнал на вході $\overline{IRQ}$ переходить на низький рівень і прапор I знято, процесор ініціює послідовність обробки переривання. Один із наслідків цього полягає в тому, що після завершення оброблення взаємодія процесора 6502 із джерелом переривання має гарантувати відсутність низького рівня сигналу на вході $\overline{IRQ}$. Якщо сигнал на вході $\overline{IRQ}$ залишиться низьким на момент виконання інструкції RTI, процесор 6502 одразу ж почне процес оброблення переривання заново. Обробка переривань, ініційованих зі входу $\overline{IRQ}$, покриває більшість рутинних операцій взаємодії між процесором 6502 і периферійними пристроями. Наприклад, у більшості комп'ютерів джерелом переривань є клавіатура.

Кожне натискання клавіші генерує переривання $\overline{IRQ}$. Під час опрацювання переривання клавіатури процесор 6502 зчитує ідентифікатор натискання клавіші через інтерфейс клавіатури та зберігає його в черзі для наступного опрацювання активним даний момент додатком. Коду обробника переривань $\overline{IRQ}$ не потрібна інформація про те, для чого будуть використовуватися дані про натискання клавіші; він просто зберігає їх для подальшого використання.

Обробка $\overline{NMI}$

Другий вхід переривань процесора 6502, $\overline{NMI}$ (Non-Maskable Interrupt), призначений для **немаскованих переривань**. Як випливає з його назви, вхід $\overline{NMI}$ не маскується за допомогою прапора I. $\overline{NMI}$ це пристрій, що заглушується за спадом сигналу.

Обробка переривань $\overline{NMI}$ виконується аналогічно обробці переривань $\overline{IRQ}$, за винятком того, що адреса процедури обробки переривань зчитується з пам'яті за адресами $FFFA–$FFFB, а прапор I не впливає на переривання цього типу.

Через те, що переривання $\overline{NMI}$ є немаскованим, воно може статися в будь-який час, включно з моментом, коли процесор 6502 заклопотаний опрацюванням переривання $\overline{IRQ}$ або навіть обробкою попереднього переривання $\overline{NMI}$.

Вхід $\overline{NMI}$ зазвичай зарезервований для подій з дуже високим пріоритетом, які не можна відкласти або пропустити. Один із можливих варіантів застосування переривань $\overline{NMI}$ це збільшення значення лічильника годинника реального часу.

Наведений нижче приклад коду обробки $\overline{NMI}$ збільшує значення 32-розрядного лічильника часу, розміщеного за адресами $10-$13, кожного разу, коли відбувається це переривання:

```
;Збільшення значення 32-розрядного лічильника часу при кожному перериванні /NMI
NMI_HANDLER:
INC $10
BNE NMI_DONE
INC $11
BNE NMI_DONE
INC $12
BNE NMI_DONE
INC $13
NMI_DONE:
RTI
```

При зверненні до апаратних сигналів у вихідному коді програми для позначення сигналу з активним низьким рівнем можна використовувати на початку пряму косу риску. У коментарі до наведеного вище прикладу коду переривання $\overline{NMI}$ позначено як /NMI.

Обробка інструкції BRK

Інструкція BRK запускає обробку, яка дуже схожа на обробку переривання $\overline{IRQ}$. Оскільки BRK – це інструкція, перед початком обробки переривання не потрібно

чекати завершення виконання поточної інструкції. При виконанні інструкції BRK адреса повернення (адреса інструкції BRK додати 2) і прапори процесора поміщаються в стек, як при переході сигналу на вході $\overline{IRQ}$ на низький рівень. Зверніть увагу, що при додаванні 2 до адреси інструкції BRK адреса повернення вказує не на перший, а на другий байт після інструкції BRK. Інструкція BRK є немаскованою: стан прапора I не впливає на її виконання.

Оброблювач інструкції BRK має ту саму адресу, що й оброблювач $\overline{IRQ}$, який знаходиться в пам'яті за адресами $FFFE-$FFFF. Оскільки інструкція BRK та $\overline{IRQ}$ використовують один і той самий обробник, визначити джерело переривання під час оброблення можна за станом прапора B. Прапор B у слові прапорів процесора, що поміщено в стек (це не прапор B у регістрі прапорів процесора P), встановлюється при виконанні інструкції BRK та скидається під час оброблення переривання $\overline{IRQ}$.

Інструкція BRK відсутня в більшості додатків процесора 6502. Традиційно ця інструкція застосовується для встановлення контрольних точок під час налагодження програми. Шляхом тимчасової заміни байту коду операції в місці, де потрібно розмістити контрольну крапку, на інструкцію BRK можна передати контроль над виконанням програмі налагодження (часто званої монітором у невеликих комп'ютерних системах), завдяки чому користувач зможе відображати й міняти вміст регістрів і клітинок пам'яті перед поновленням виконання.

У наступному прикладі коду реалізовано мінімальний обробник, який розрізняє переривання $\overline{IRQ}$. та інструкції BRK. У ньому як місце тимчасового зберігання використовується адреса пам'яті $14:

```
; Обробка переривань /IRQ та інструкцій BRK
IRQ_BRK_HANDLER:
; Збереження вмісту регістру A
STA $14
; Витяг прапорців процесора зі стека в регістр A
PLA
PHA
; Перевірка встановлення біта B в слові прапорців в стеку
AND $10 ; $10 вибирає біт B
; Якщо результат відмінний від нуля, то біт B був встановлений:
; обробка для інструкції BRK
BNE BRK_INSTR
; біт B не був встановлений: додайте сюди обробку для переривання /IRQ
; • • •
JMP IRQ_DONE
```

```
BRK_INSTR:
; Додайте сюди обробку для інструкції BRK
; • • •
IRQ_DONE:
; Відновлення регістра A і повернення
LDA $14
RTI
```

Цей приклад показав, як розрізняти переривання, ініційовані зі входу $\overline{IRQ}$ процесора та інструкцій BRK, в архітектурі процесора 6502. У складніших процесорах, включно з тими, які ми обговоримо в наступних розділах, для кожного вхідного сигналу, що ініціює переривання, передбачено унікальні вектори переривань (початкові адреси обробників переривань). Ці архітектури також містять широку підтримку функцій налагодження, таких як встановлення контрольних точок у місцях розташування зазначених інструкцій.

У попередніх розділах було представлено категорії інструкції в архітектурі процесора 6502 і подано стислий опис кожної інструкції в межах цих категорій. Процесор 6502 набагато ширший, ніж сучасні 32- і 64-розрядні процесори, однак у цьому обговоренні було представлено найпоширеніші типи інструкцій і режимів адресації, які використовуються навіть у найскладніших сучасних процесорах, включно з інструкціями, що підтримують універсальну концепцію оброблення переривань.

У наступному розділі буде представлено основи обробки вводу-виводу, що забезпечує передачу даних між процесором і периферійними пристроями.

Операції введення-виведення

Метою частини архітектури процесора, що відповідає за операції введення-виведення, є ефективне передавання даних між зовнішніми периферійними пристроями та пам’яттю системи. Операції введення переносять дані із зовнішнього світу в пам’ять, а операції виведення відправляють дані з пам’яті зовнішнім адресатам.

Формат даних на зовнішній стороні інтерфейсу введення-виведення може бути найрізноманітнішим. Ось кілька прикладів зовнішнього представлення даних вводу-виводу комп’ютера:

- сигнали, що передаються по відеокабелю, приєднаному до монітора;
- коливання напруги на проводах кабелю Ethernet;
- картина ліній магнітного поля на поверхні диска;
- звукові хвилі, що генеруються динаміками комп’ютера.

Незалежно від того, якої форми дані набувають поза комп'ютером, з'єднання будь-якого пристрою введення-виведення з процесором повинно відповідати архітекторі введення-виведення процесора, а пристрій введення-виведення має бути сумісним з будь-якими іншими пристроями введення-виведення, що є присутніми в комп'ютерній системі.

Для взаємодії з пристроями введення-виведення процесор використовує категорії інструкцій, режими адресації та методи обробки переривань, описані раніше в цьому розділі. Відмінність полягає в тому, що замість читання і запису даних у системну пам'ять ці інструкції здійснюють зчитування і запис даних у розділи пам'яті, які використовуються для обміну даними з пристроями введення-виведення.

У сучасних процесорах для доступу до пристроїв введення-виведення застосовують два основні підходи: **введення-виведення з розподілом пам'яті** та **введення-виведення з розподілом за портами**. При введенні виведенні з розподілом пам'яті пристроям введення-виведення виділяються певні частини системного адресного простору. Процесор звертається до периферійних пристроїв за заздалегідь визначеними адресами за допомогою тих самих інструкцій і режимів адресації, що використовуються для читання і запису в системну пам'ять. У процесорі 6502 для зв'язку з периферійними пристроями прийнято підхід із розподілом пам'яті.

У процесорах, що використовують введення-виведення з розподілом за портами, для виконання операцій введення-виведення реалізовано окрему категорію інструкцій. При цьому для пристроїв введення-виведення з розподілом за портами виділяють адресний простір, що не належить до системної пам'яті. Як адреси пристроям введення-виведення призначаються **номери портів**. В архітектурі x86 використовується введення-виведення з розподілом по портах.

Одним із недоліків введення-виведення з розподілом пам'яті є необхідність виділення для пристроїв введення-виведення частини системного адресного простору, що веде до зменшення максимального об'єму пам'яті, який може бути встановлений у комп'ютерній системі. Недолік введення-виведення з розподілом по портах – вимога реалізації в процесорі додаткових інструкцій для виконання операцій введення-виведення.

У деяких реалізаціях введення-виведення з розподілом по портах передбачається надання додаткових апаратних сигналів, що позначають звернення до пристрою введення-виведення, а не до системної пам'яті. Представивши цей сигнал у вигляді селектора (який фактично стає ще одним бітом адреси), одні й ті самі адресні рядки можна використовувати для доступу і до пам'яті, і до пристроїв введення-виведення. Як альтернативне рішення для виконання операцій введення-виведення з розподілом по портах у деяких більш високопродуктивних процесорах реалізують повністю виділену шину. Така архітектура дає змогу одночасно виконувати операції введення-виведення та доступу до пам'яті.

У найпростішому підході до введення-виведення процесор сам обробляє всі етапи операцій введення-виведення, використовуючи інструкції для передачі даних між пам’яттю і пристроєм введення-виведення. У процесорах зі складнішою архітектонікою передбачено архітектурні функції для прискорення повторюваних операцій введення-виведення. Ми обговоримо три методи виконання операцій введення-виведення з різним ступенем використання процесора: програмоване введення-виведення, введення-виведення з керуванням за перериваннями і прямий доступ до пам’яті.

Програмоване введення-виведення

Програмоване введення-виведення означає, що процесор виконує кожний крок операції передачі даних введення-виведення з використанням програмних інструкцій. Розглянемо клавіатуру, яка представлена процесору у вигляді двох відображених у пам’яті однобайтових адрес в адресній області введення-виведення процесора. Один із цих байтів містить інформацію про стан, зокрема біт, що вказує, коли було натиснута клавіатура. Другий байт містить значення клавіші, яка була натиснута.

При кожному натисканні клавіші встановлюється біт стану "натиснута клавіша". У разі використання програмованого вводу-виводу процесор має періодично зчитувати регістр стану клавіатури, щоб визначити, чи була натиснута клавіша. Якщо біт стану вказує на натискання клавіші, процесор зчитує регістр даних клавіатури, при цьому біт стану "натиснуто клавішу" скидає до наступного натискання клавіші.

Якщо в регістрі даних клавіатури одночасно може зберігатися інформація тільки про одну клавішу, то операцію перевірки стану клавіатури необхідно виконувати досить часто, щоб жодне натискання клавіш не губилося, навіть якщо за клавіатурою працює користувач, який дуже швидко рухається. У результаті процесор повинен витрачати значну кількість свого часу на перевірку натискань клавіш. За відсутності швидкого введення тексту більшість із цих перевірок будуть безрезультатними.

Очевидно, що програмоване введення-виведення не є високоефективним методом для розв’язання задач загального призначення. За своєю суттю це схоже на перевірку вашого телефону кожні кілька секунд, щоб дізнатися, чи дзвонить вам хто-небудь.

Програмоване введення-виведення має сенс в окремих ситуаціях. Наприклад, одноразове налаштування периферійного пристрою під час запуску системи є цілком розумним застосуванням цього методу.

Введення-виведення з керуванням за перериваннями

Пристрій введення-виведення може використовувати переривання для сповіщення процесора про необхідність виконання дій. У разі простого інтерфейсу клавіатури замість встановлення біта в регістрі станів периферійний пристрій може встановлювати на лінії $\overline{IRQ}$ процесора 6502 низький рівень для виклику переривання

при кожному натисканні клавіші. Такий підхід дає змогу звільнити процесор від постійних перевірок натискань клавіш, щоб не відволікати його від виконання інших операцій. Процесор звертатиме увагу на інтерфейс клавіатури тільки тоді, коли необхідно виконати роботу, на що вказує переривання. Використання переривань для запуску операцій введення-виведення аналогічно додаванню дзвінка в телефон, який ми повинні були перевіряти наявність вхідних викликів кожні кілька секунд під час використання програмованого введення-виведення.

У процесорі 6502 для операцій введення-виведення передбачено єдиний маскований **вхідний сигнал виклику переривань** ($\overline{IRQ}$). Комп'ютерні системи зазвичай містять кілька джерел переривань введення-виведення. Це трохи ускладнює завдання обслуговування переривання в 6502, оскільки процесор повинен спочатку визначити, який периферійний пристрій ініціював переривання, перш ніж він зможе почати передачу даних.

Для того щоб визначити джерело переривання, обробник переривань повинен опитати кожен пристрій, що підтримує такі переривання. У разі інтерфейсу клавіатури ця операція опитування складається зі зчитування регістра стану клавіатури, щоб визначити, чи встановлений біт, який вказує на те, що відбулося натискання клавіші. Щойно процесор ідентифікує пристрій, відповідальний за переривання, він переходить до коду, який взаємодіє з пристроєм, для виконання запитуваної задачі введення-виведення. У випадку інтерфейсу клавіатури код опрацювання виконує читання регістра даних клавіатури та очищення біта стану, що вказує на натискання клавіші, у результаті чого вхідний сигнал $\overline{IRQ}$ вимикається.

Переривання від зовнішніх пристроїв є асинхронними подіями – це означає, що вони можуть статися в будь-який момент часу. Проектування комп'ютерної системи вимагає ретельного аналізу навіть найменш імовірних можливостей формування переривань, наприклад під час запуску системи або під час опрацювання інших переривань. Переривання від кількох пристроїв можуть надходити одночасно або майже одночасно і у випадковому порядку. Апаратні схеми опрацювання переривань і код обслуговування переривань мають забезпечувати виявлення й опрацювання всіх переривань незалежно від їхніх часових характеристик.

Керування введенням-виведенням на основі переривань позбавляє процесор від необхідності періодично перевіряти пристрої введення-виведення, щоб визначити, чи потрібне виконання якої-небудь дії. Однак обробка переривань може зайняти значний час процесора, якщо воно пов'язане з передачею великого блоку даних. Подібна ситуація часто виникає під час таких операцій, як читання з диску або запис на нього. Наступний метод введення-виведення, який ми обговоримо, усуває потребу в залученні процесора до роботи з передачі таких великих блоків даних.

Прямий доступ до пам’яті

Прямий доступ до пам’яті (direct memory access, DMA) дає змогу периферійним пристроям отримувати доступ до системної пам’яті незалежно від процесора. У разі використання DMA для передання блока даних процесор налаштовує цю операцію, передаючи в контролер DMA початкову адресу переміщуваного блока даних, довжину блока й адресу місця призначення. Після запуску процедури DMA процесор може продовжити роботу з виконання інших операцій. Після завершення операції контролер DMA генерує переривання, щоб повідомити процесору про те, що передача даних завершена.

У комп’ютерній системі контролер DMA можна реалізувати у вигляді окремої інтегральної схеми, керованої процесором, або архітектура процесора може містити один або кілька вбудованих DMA-контролерів.

Пристрої введення-виведення, які переміщують значні об’єми даних, такі як дискові накопичувачі, звукові карти, відеокарти та мережеві інтерфейси, зазвичай використовують DMA для ефективного передавання даних до системної пам’яті та з неї. Технологію DMA також використовують для передавання блоків даних у системній пам’яті.

Архітектура процесора 6502 не підтримує операції DMA, але оригінальний IBM PC містив у собі DMA-контролер. Архітектура майже будь-якого 32- або 64-розрядного процесора забезпечує широку підтримку операцій DMA.

DMA – один із багатьох методів, які підвищують продуктивність комп’ютерної системи за рахунок прискорення повторюваних операцій. У главах 5 і 9 ми розглядаємо кілька прикладів використання DMA для прискорення операцій введення-виведення.

Резюме

У цьому розділі було описано основні функціональні блоки простого процесора: блок керування, арифметико-логічний пристрій і регістри. За описами цих компонентів пішов огляд інструкцій і режимів адресації процесора. З метою демонстрації різноманітності та корисності інструкцій, доступних у відносно простій процесорній архітектурі, було розглянуто категорії інструкцій, реалізовані в процесорі 6502.

Також у контексті архітектури процесора 6502 було представлено та продемонстровано концепції, пов’язані з обробленням переривань. Розділ завершився оглядом найпоширеніших архітектурних підходів до керування операціями введення-виведення (введення-виведення з розподілом пам’яті та введення-виведення з розподілом по портах) та основних режимів виконання операцій введення-виведення в комп’ютерній системі (програмований введення-виведення, введення-виведення на основі переривання та прямий доступ до пам’яті).

Після завершення цього розділу ви повинні мати загальне уявлення про функціональні блоки процесора, обробку інструкцій і переривань, а також про операційні блоки процесора. раціях введення-виведення. Ця інформація є основою для наступного розділу, у якому розглядається архітектура на рівні комп'ютерної системи.

Вправи

1. Розглянемо додавання двох 8-бітних чисел зі знаком (тобто чисел у діапазоні від -128 до +127), де один операнд додатний, а інший – від'ємний. Чи існує якась пара 8-бітних чисел із різними знаками, сума яких виходить за межі діапазону від -128 до +127? Така ситуація буде являти собою знакове переповнення. Примітка: ми розглядаємо тут тільки додавання, тому що, як ми бачили, віднімання в архітектурі процесора 6502 – це те ж саме, що і додавання, але з інверсією бітів правого операнда.
2. Якщо відповідь на вправу 1 – "ні", це означає, що єдиний спосіб створити знакове переповнення – скласти два числа з однаковими знаками. Якщо відбувається переповнення, що ви можете сказати про результат виконання операції "виключне АБО" між старшим бітом кожного операнда і старшим бітом результату? Іншими словами, яким буде результат виразів left(7) XOR result(7) і right(7) XOR result(7)? У цих виразах (7) вказує на біт 7 – старший біт.
3. Подивіться на лістинг VHDL у розд. "Арифметико-логічний пристрій" у цьому розділі та визначте, чи є логіка встановлення та скидання прапора V коректною для операцій додавання та віднімання. Перевірте результати додавання 126 + 1, 127 + 1, -127 + (-1) і -128 + (-1). При пересиланні даних через середовище передавання, схильне до помилок, щоб визначити, чи були які-небудь біти даних втрачені або спотворені під час передавання, зазвичай використовується контрольна сума. Вона, як правило, додається до переданого запису даних. В одному з алгоритмів розрахунку контрольної суми використовуються такі кроки:
   1) додавання всіх байтів у записі даних зі збереженням тільки молодших 8 біт суми;
   2) розподіл контрольної суми, яка являє собою доповнення до двох цієї 8-бітної суми;
   3) додавання байта контрольної суми до запису даних.

   Але після отримання блоку даних з доданою контрольною сумою процесор може визначити правильність контрольної суми простим додаванням усіх байтів у запису, включно з контрольною сумою. Контрольна сума є правильною, якщо молодші 8 біт суми дорівнюють нулю. Реалізуйте цей алгоритм контрольної суми, використовуючи мову асемблера 6502. Байти даних починаються з комірки пам'яті за адресами $10-$11, а кількість байтів (включно з байтом контрольної суми) надається як вхідні дані в регістрі X. Встановіть для регістра A значення 1, якщо контрольна сума є правильною, і значення 0 в іншому випадку.

4. Помістіть код перевірки контрольної суми із *вправи 4* у позначену підпрограму, яку можна викликати за допомогою інструкції JCR і яка завершується інструкцією RTS.
5. Назвіть і виконайте набір тестів для перевірки правильності роботи підпрограми перевірки контрольної суми, яку ви створили у *вправах 4-5*. Яка довжина найкоротшого блоку даних, для якого ваш код може виконати перевірку контрольної суми? Яка довжина найдовшого блоку даних?

4

Компоненти комп’ютерної системи

У цьому розділі представлено більш низькорівневі компоненти, що використовуються при побудові комп’ютерних систем. Ми почнемо з транзистора на основі структури **метал-оксид-напівпровідник** (МОН), який знайшов широке застосування в схемах пам’яті, а також практично у всіх інших сучасних цифрових пристроях. Ми розглянемо будову комп’ютерної пам’яті на основі МОН-транзисторів та її інтерфейс із процесором. Ми познайомимося із сучасними комп’ютерними інтерфейсами введення-виведення, приділяючи особливу увагу високошвидкісному послідовному зв’язку усередині комп’ютера, а також передаванню даних через кабельні з’єднання зовнішнім компонентам. Буде розглянуто функціональні вимоги до пристроїв системного введення-виведення, що включають дисплей, мережевий інтерфейс, клавішу та мишу. Розділ закінчується наочним прикладом технічних характеристик материнської плати сучасного комп’ютера.

Після ознайомлення з даним розділом ви отримаєте чітке уявлення про апаратні компоненти сучасних комп’ютерних систем – від їхніх технічних характеристик до схемного рівня. Ви знатимете, як реалізовано системну пам’ять, включно з основами кешування. Розумітимете механізми ефективних операцій введення-виведення та використання інтерфейсу **Universal Serial Bus** (USB) для підключення клавіатури, миші та інших пристроїв введення-виведення. Ви отримаєте уявлення про мережевий інтерфейс і познайомитеся з основними технічними характеристиками материнської плати сучасного комп’ютера.

У цьому розділі буде розглянуто такі теми:

- підсистема пам’яті;
- знайомство з польовими МОН-транзисторами;
- побудова схем динамічної пам’яті за допомогою польових МОН-транзисторів;
- підсистема введення-виведення;
- графічні дисплеї;
- мережевий інтерфейс;
- клавіатура та миша;
- технічні характеристики сучасних комп’ютерних систем.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-Organizайon-Second-Edition.**

Підсистема пам’яті

Підсистема пам’яті це адресована послідовність комірок пам’яті, що містять інструкції та дані для використання процесором під час виконання програм. Сучасні комп’ютерні системи та цифрові пристрої часто містять понад мільярд 8-бітних комірок в основній пам’яті, кожна з яких може бути незалежно зчитана та записана процесором.

Як було показано в *главі 1*, конструкція аналітичної машини Беббіджа як засобу зберігання даних для обчислень містила в собі набір осей, на кожній з яких були встановлені 40 коліс, що відображають десяткові розряди. Зчитування даних з осі було руйнівною операцією, що приводила до скидання цифр на кожному з коліс осі після завершення зчитування. Це був повністю механічний метод зберігання даних.

З 1950-х до 1970-х років переважною технологією реалізації цифрової комп’ютерної пам’яті був магнітний сердечник. Один біт пам’яті на магнітних сердечниках зберігається в невеликому тороїдальному (у формі пончика) керамічному постійному магніті. Сердечники набору, що складає масив пам’яті, розташовані у вузлах прямокутної сітки, утвореної горизонтальними і вертикальними з’єднувальними проводами. Запис у бітову комірку полягає в подачі по дротах, які проходять через серцевину, достатнього струму, щоб можна було змінити полярність його магнітного поля. Біт 0 може бути визначений як циркулювання магнітного потоку всередині сердечника за годинниковою стрілкою, а біт 1 – як циркулювання потоку проти годинникової стрілки.

Зчитування біта з пам’яті на магнітних осердях полягає у спробі встановити для біта полярність 0 і спостереженні за електричним відгуком. Якщо обраний сердечник уже містить біт 0, то відгуку не буде. Якщо сердечник містить 1, то при зміні полярності виникає помітний імпульс напруги. Як і в аналітичній схемі Беббіджа, операція читання з пам’яті на магнітних осердях є руйнівною. Але після зчитування бітового значення 1 з пам’яті необхідно виконати наступний запис, щоб відновити стан біта.

Пам’ять на магнітних сердечниках є енергонезалежною: після вимкнення живлення її вміст зберігається протягом невизначеного терміну. Вона також має властивості, які роблять її цінною в таких сферах застосування, як космічні апарати, де важлива стійкість до радіації. Відомо, що комп’ютери космічних кораблів системи "Спейс Шаттл" використовували пам’ять на магнітних сердечниках до кінця 1980-х років.

У сучасних комп’ютерних системах споживчого та бізнес-класу для основної системної пам’яті майже винятково використовують схеми динамічної пам’яті довільного доступу (dynamic random access memory, DRAM) на основі польових МОН-транзисторів (metal-oxide-semiconductor field-effect transistor, MOSFET). У наступному розділі представлено особливості польових МОН-транзисторів.

Знайомство з польовими МОН-транзисторами

У розділі 2 описано n-p-n-транзистор, що належить до типу **біполярних транзисторів** (bipolar junction transistor, BJT). Транзистор типу n-p-n називається біполярним, оскільки для його функціонування використовуються як позитивні (p), так і (n) носії заряду.

У напівпровідниках носіями негативного заряду служать електрони. У роботі напівпровідника не беруть участі фізичні частинки з позитивним зарядом. Замість цього за відсутності зазвичай присутнього електрона атом проявляє ті самі властивості, що й позитивно заряджена частинка. Ці відсутні називаються **дірками**. У біполярних транзисторах дірки виконують носіїв позитивного заряду.

Концепція дірок є настільки фундаментальною для роботи напівпровідників, що Вільям Шоклі (William Shockley), один із винахідників транзистора, написав книжку під назвою "Електрони і дірки в напівпровідниках" ("Electrons and Holes in Semiconductors"), яку було опубліковано в 1950 р. Далі ми розглянемо поведінку позитивних і негативних носіїв заряду в уніполярних транзисторах.

На відміну від біполярних транзисторів, **уніполярний транзистор** покладається тільки на один із двох типів носіїв заряду. **Польовий транзистор на основі структури метал-оксид-напівпровідник** (**польовий МОН-транзистор** – metal-oxide-semiconductor field-effect transistor, MOSFET) – це уніполярний транзистор, придатний для використання як цифровий перемикальний елемент. Як і транзистор типу n-p-n, польовий МОН-транзистор являє собою триполюсний пристрій із керуючим входом для ввімкнення та вимкнення струму, що протікає через два інші виводи. Виводи польового МОН-транзистора називають **затвором, стоком** і **витоком**. Вивід затвора керує протіканням струму між виводами стоку і витоку.

Польові МОН-транзистори відносяться до категорії пристроїв, що працюють у **режимі збагачення носіями** або в **режимі збіднення носіями**. МОН- транзистор у режимі збагачення носіями блокує протікання струму між стоком і витоком, коли напруга на затворі дорівнює нулю, і дозволяє протікання струму, коли напруга на затворі перевищує порогову напругу. МОН-транзистор у режимі збіднення носіями функціонує протилежним чином, блокуючи протікання струму за високої напруги на затворі та дозволяючи його протікання, коли напруга на затворі дорівнює нулю.

На рис. 4.1 показано схематичне представлення n-канального МОН-транзистора в режимі збагачення носіями.

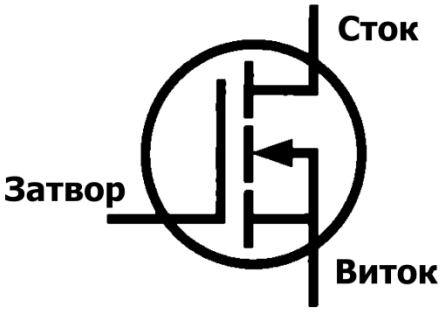


Рис 4.1. *n-канальний МОН-транзистор у режимі збагачення носіями*

Для наших цілей n-канальний МОН-транзистор у режимі збагачення носіями функціонує як перемикач: коли напруга на затворі відносно витоку низька (нижча за порогову напругу), між стоком і витоком виникає дуже високий опір. Коли напруга на затворі відносно витоку висока (вища за порогову напругу), опір між цими виводами дуже малий. Буква n у позначенні "n-канальний" пов’язана з каналом у кремнії, який був легований для збільшення числа електронів (носіїв негативного заряду).

Поведінка польового МОН-транзистора нагадує роботу n-p-n-транзистора, розглянутого в *розділі 2*. Однак є важлива відмінність: МОН-транзистор – це пристрій, керований напругою, тоді як n-p-n -транзистор керується струмом. У разі використання n-p-n-транзистора як перемикача, щоб струм протікав між його емітером і колектором, через базу має протікати невеликий, але постійний струм. У разі ж МОН-транзистора для протікання струму між стоком і витоком потрібно лише докласти між затвором і витоком напругу, величина якої вища за порогове значення. Для утримання такого перемикача у відкритому стані протікання струму через вхід затвора практично не потрібне. Завдяки цьому МОН-транзистор споживає значно менше енергії, ніж n-p-n -транзистор, що виконує еквівалентну цифрову функцію.

Мохаммед Аталла (Mohamed Atalla) і Давон Канг (Dawon Kahng) з Bell Telephone Laboratories винайшли МОН-транзистор у 1959 р. І тільки на початку 1970-х років виробничі процеси стали достатньо розвиненими, щоб забезпечити надійне виготовлення інтегральних схем на основі МОН-структури. Відтоді МОН-транзистор є найпоширенішим типом транзисторів, що використовуються в інтегральних схемах. Згідно з оцінками, у 2018 р. було виготовлено 13 секстильйонів транзисторів (секстильйон — це число, що позначається одиницею з 21 нулем), 99,9% з яких були МОН-транзисторами. МОН-транзистор є найбільш часто виробленим пристроєм в історії людства.

p-канальний МОН-транзистор у режимі збагачення носіями подібний до n-канального МОН-транзистора в режимі збагачення носіями, за винятком того, що він демонструє протилежну реакцію на напругу між затвором і виводом витоку: якщо напруга на виводі затвора нижча, ніж на виводі витоку, і різниця між ними перевищує порогову напругу, то струм протікає між стоком і витоком, а якщо напруга між затвором і витоком менша за це порогове значення, то протікання струму між стоком і витоком блокується. Буква p в позначенні "p-канальний" вказує на легування каналу, яке

збільшує кількість дірок (носіїв позитивного заряду). На рис. 4.2 показано схематичне представлення p-канального МОН-транзистора в режимі збагачення носіями.

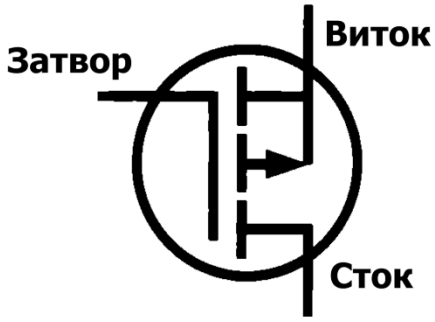


Рис 4.2. *p-канальний МОН-транзистор у режимі збагачення носіями*

Для того щоб розрізняти схематичні уявлення n- і p-канальних МОН-транзисторів, зверніть увагу, що в обох конфігураціях вивід витоку приєднано до центрального з трьох внутрішніх з'єднань. У p-канальному МОН-транзисторі стрілка на цьому з'єднанні вказує на затвор, а в p-канальному МОН-транзисторі – у напрямку від затвора.

Як n-канальні, так і p-канальні МОН-транзистори в режимі збагачення носіями можна вважати нормально розімкненими перемикачами. Це означає, що вони не проводять струм, коли напруга між затвором і витоком мала. Як n-канальні, так і p-канальні МОН-транзистори також доступні в конфігурації, що забезпечує роботу в режимі збіднення носіями, що дає змогу їм функціонувати як нормально замкнуті перемикачі. Струм протікає через МОН-транзистори, що працюють у режимі збіднення носіями, коли напруга між затвором і витоком є низькою, але не надто високою.

Для виконання логічних функцій n- і p-канальний МОН-транзистори часто використовують у парі. Пристрій, що містить таку пару МОН-транзисторів, називається інтегральною схемою на основі **комплементарної МОН-структури** (КМОН-схемою). Назва КМОН відображає той факт, що транзистори в кожній парі функціонують протилежним або взаємодоповнюючим чином. Поза періодами перемикання КМОН-схеми майже не споживають енергію, оскільки входи затворів не потребують практично ніякого струму. Структуру КМОН-схеми розробили Чі-Тан Сах (Chih-Tang Sah) і Френк Ванласс (Frank Wanlass) з компанії Fairchild Semiconductor у 1963 р.

На рис. 4.3 показано схему вентиля НЕ, у якій n-p-n-транзистор з *розділу 2* був замінений парою комплементарних польових МОН-транзисторів.

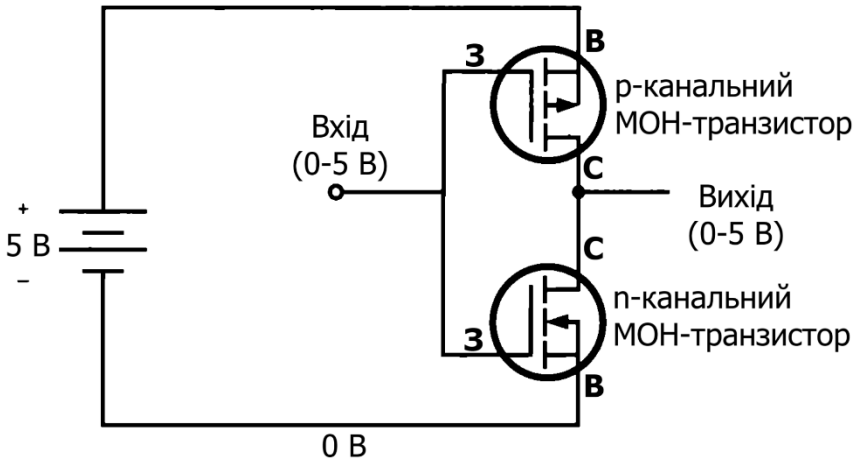


Рис 4.3. *КМОН-схема вентиля НЕ*

Коли сигнал **Вхід** має низький рівень (поблизу 0 В), нижній n-канальний МОН-транзистор вимикається, а верхній p-канальний МОН-транзистор вмикається. У результаті **Вихід** підключається до позитивного полюса джерела напруги, що викликає збільшення рівня сигналу **Вихід** приблизно до 5 В. Коли сигнал **Вхід** має високий рівень, верхній МОН-транзистор вимкнений, а нижній МОН-транзистор увімкнений, що спричиняє зниження сигналу **Вихід** майже до 0 В. Сигнал **Вихід** завжди є інвертованим щодо сигналу **Вхід**, що відповідає поведінці вентиля НЕ.

Сьогодні практично всі цифрові інтегральні схеми високої щільності засновані на технології КМОП. Крім виконання логічних функцій, польовий МОН-транзистор є ключовим компонентом сучасної архітектури схем оперативної пам’яті (random access memory, RAM). У наступному розділі обговорюється використання польових МОН-транзисторів у схемах пам’яті.

Побудова схем динамічної пам’яті за допомогою польових МОН-транзисторів

Один біт у стандартній інтегральній схемі динамічної пам’яті дозвільного доступу (DRAM) складається з двох схемних елементів: МОН-транзистора і конденсатора. У наступному розділі подано короткий вступ в електричні характеристики конденсаторів.

Конденсатор

Конденсатор – це пасивний схемний елемент із двома виводами, здатний накопичувати енергію. Енергія надходить у конденсатор і залишає його у вигляді електричного струму. Напруга на виводах конденсатора пропорційна кількості електричної енергії, що міститься в ньому. Продовжуючи аналогію з гідравлічною системою, описаною в розділі 2, уявіть, що конденсатор – це повітряна кулька, приєднана до бічної сторони труби, яка веде до водопровідного крана. Тиск води в трубі змушує кульку надуватися, наповнюючи її водою з труби. Припустимо, що це

міцна повітряна кулька, і під час надування вона розтягується, збільшуючи внутрішній тиск. Кулька наповнюється доти, доки тиск у ній не зрівняється з тиском у трубі, після чого перестає наповнюватися.

Якщо повністю відкрити кран на кінці труби, то внаслідок випуску води тиск у трубі зменшиться. Вода з кульки витікатиме назад у трубу доти, доки тиск у кульці знову не зрівняється з тиском у трубі.

Гідравлічні пристрої, так звані **амортизаторами гідравлічних ударів**, працюють саме в такий спосіб, розв'язуючи проблему труб, які видають звуки, що стукають, під час відкривання і закривання водопровідних кранів. Амортизатор гідравлічних ударів демонструє поведінку, подібну до розтягування повітряної кульки, згладжуючи різкі зміни тиску води, що виникають під час відчинення та зачинення кранів.

Кількість електричної енергії, що міститься в конденсаторі, аналогічна кількості води в повітряній кульці. Напруга на конденсаторі аналогічна тиску всередині повітряної кульки, що виникає під час її розтягування.

Електричний конденсатор можна виготовити з двох паралельних металевих пластин, розділених ізолювальним матеріалом, наприклад повітрям. Виходи конденсатора приєднані до кожної з пластин. Відношення кількості запасеної електричної енергії до напруги на конденсаторі називається ємністю. Вона залежить від розміру паралельних пластин, відстані між ними і типу матеріалу, що використовується як ізолятор. Ємність конденсатора аналогічна розміру повітряної кульки в прикладі з гідравлічною системою. Конденсатор із більшою ємністю відповідає кульці більшого розміру. Для наповнення великої повітряної кульки до заданого тиску потрібно більше води, ніж для наповнення маленької.

Схематичне позначення конденсатора показано на рис. 4.4



Рис 4.4. *Схематичне позначення конденсатора*

Дві горизонтальні лінії із зазором між ними відображають конструкцію конденсатора з металевими пластинами, описану в цьому розділі. Одиницею вимірювання ємності є **фарад**, названий на честь англійського вченого Майкла Фарадея, який, окрім багатьох інших своїх здобутків, винайшов електродвигун.

Бітова комірка динамічної пам'яті

Бітовий осередок динамічної пам'яті (dynamic random access memory, DRAM) – це доступне для читання і запису місце зберігання одного біта даних. Модуль пам'яті DRAM у сучасному комп'ютері або телефоні містить мільярди бітових комірок. Один

біт у схемі DRAM складається з МОН-транзистора і конденсатора, розташованих так, як показано на рис. 4.5.

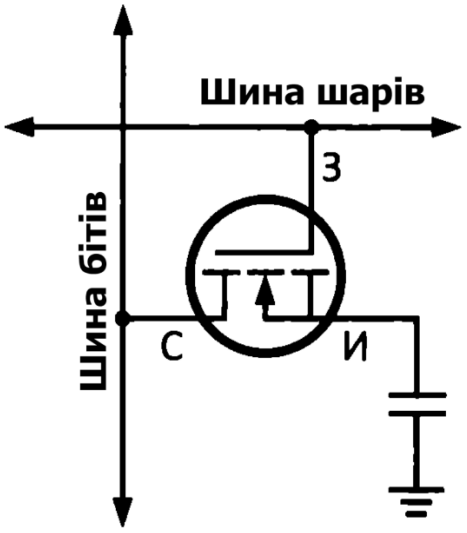


Рис 4.5. *Схема бітової комірки DRAM*

На цьому малюнку символ із трьома горизонтальними лініями в правому нижньому кутку є символом землі – стандартного представлення опорної напруги 0 В, яке ми використовували в попередніх схемах, таких як показана на рис. 4.3. Шина слів і шина бітів – це дроти, що використовуються для з'єднання окремих бітових комірок у сітку.

Ця одно бітова комірка повторюється в прямокутній сітці, утворюючи повний банк пам'яті DRAM. На рис. 4.6 показано конфігурацію 16-бітного банку пам'яті DRAM, що складається з 4 слів по 4 біти кожне.

Реальні банки пам'яті DRAM містять набагато більшу кількість бітів, ніж показана тут проста схема. Типові пристрої DRAM мають розмір слова 8 біт, а не 4 біти, позначені на цьому малюнку як B_0-B_3. Це означає, що схема DRAM дає змогу паралельно зберігати або витягувати 8 біт.

Кількість екземплярів бітових комірок на шині слів у реальному масиві банків DRAM є цілим числом, кратним розміру слова пристрою. Великі модулі DRAM, використовувані в персональних комп'ютерах, містять багато слів на кожній шині слів. Наприклад, мікросхема DRAM з 8- бітовими словами і 1024 словами в рядку містить 8192 біти в рядку, при цьому всі виводи затворів МОН-транзисторів уздовж рядка керуються одним сигналом по шині слів. Ці пристрої містять додаткову логіку мультиплексора для вибору конкретного запитуваного процесором слова з безлічі слів у ряду, обраному активною шиною слів.

Вертикальний вимір банку пам'яті DRAM складається з реплікованих копій рядів комірок з однією шиною слів, що керує кожним рядом. Шина слів з'єднує всі бітові комірки по горизонталі, а сигнали шини бітів – комірки у всіх рядках по вертикалі.

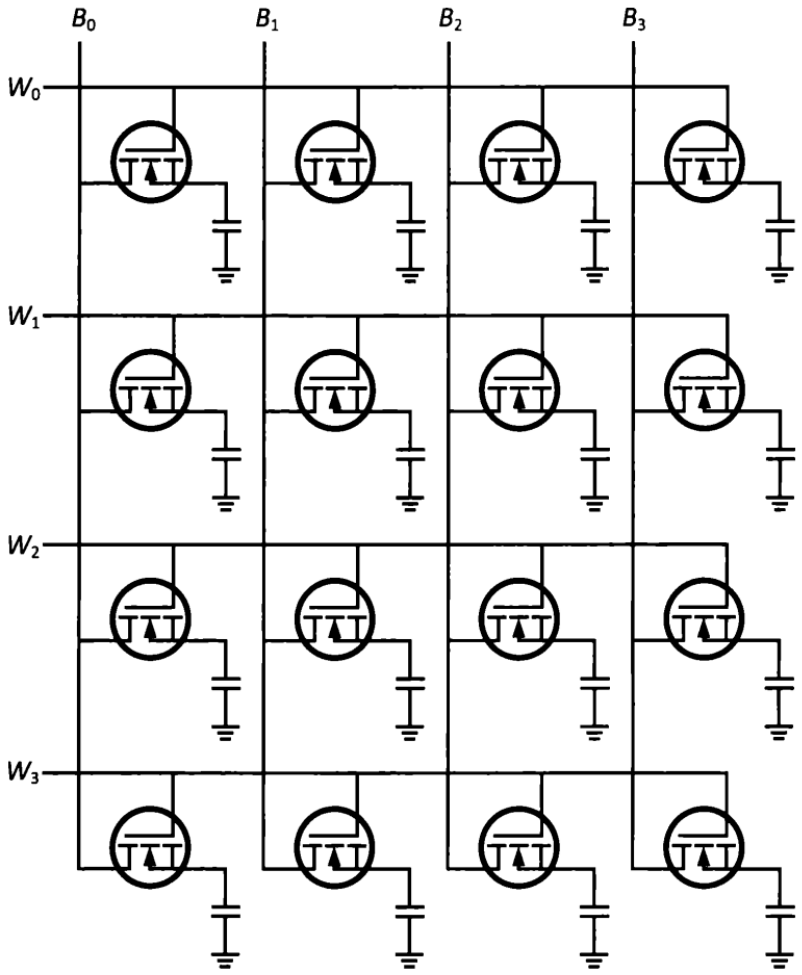


Рис 4.6. *Організація банку пам'яті DRAM*

Стан кожного біта пам'яті зберігається в конденсаторі комірки. Низька напруга на конденсаторі відображає значення 0, а висока напруга – значення 1. У контексті пристроїв пам'яті DRAM DDR5 низька напруга – близько 0 В, а висока – близько 1,1 В.

На шині слів для кожного ряду більшу частину часу підтримується низька напруга. Це забезпечує перебування МОН-транзисторів у вимкненому стані, за якого зберігається стан конденсатора. Коли настає час прочитати слово (фактично – цілий ряд) з пристрою DRAM, схема адресації вибирає відповідну шину слів і подає на неї напругу високого рівня, а на всіх інших шинах слів у банку зберігається низька напруга. При цьому МОН-транзистори в кожній бітовій комірці на активній шині слів відкриваються, подаючи напругу конденсаторів цих комірок на під'єднані шини бітів. Шини бітів для комірок у стані 1 (високий рівень) матимуть більш високу напругу на всіх шинах слів, ніж шини для комірок у стані 0 (низький рівень). Напруги на шинах бітів зчитуються схемою пристрою DRAM і фіксуються у вихідному регістрі мікросхеми.

Запис у слово DRAM починається з встановлення високого рівня на обраній шині слів так само, як і під час читання слова. Замість вимірювання напруг на шинах бітів пристрій DRAM подає на кожну бітову лінію напругу, яку потрібно записати в кожний осередок, – 0 або 1,1 В, залежно від того, яке значення має отримати біт даних – 0 або

1. Як і під час наповнення або спорожнення повітряної кульки, кожному конденсатору потрібен певний час, щоб зарядитися або розрядитися до напруги, що діє на шині бітів, до якої він приєднаний. Але після закінчення цієї затримки на шину слів подається низький рівень, щоб закрити МОН-транзистори і зафіксувати нові рівні заряду конденсаторів.

Технологія схем DRAM ускладняється тим фактом, що конденсаторам властивий витік. Після заряджання конденсатора до ненульової напруги заряд з часом вщухає, зменшуючи напругу на конденсаторі. Через це вміст кожної комірки необхідно періодично оновлювати.

Операція оновлення зчитує значення кожної комірки і потім записує його назад у комірку. При цьому конденсатор перезаряджається до "повного" рівня напруги, якщо він представляє значення 1, або розряджається майже до 0 В, якщо він представляє значення 0. Типовий інтервал оновлення для сучасних пристроїв DRAM становить 64 мілісекунди. Оновлення DRAM відбувається безперервно, рядок за рядком, у фоновому режимі під час роботи системи і синхронізується з доступом процесора до пам’яті, щоб уникнути конфліктів. Під час цього процесу спостерігається невелике зниження продуктивності, яке виникає, коли доступ процесора до пам’яті затримується через виконання оновлення.

Необхідність періодичного оновлення значно ускладнює конструкцію систем, що використовують пристрої пам’яті DRAM, проте переваги зберігання стану біта за допомогою всього лише одного транзистора й одного конденсатора настільки великі, що DRAM витіснила всі альтернативи як переважну технологію для основної пам’яті в комп’ютерних системах споживчого, комерційного та наукового призначення.

У наступному розділі буде розглянуто архітектуру поточного покоління технології DRAM – DDR5.

SDRAM DDR5

Компанія Intel випустила першу комерційну інтегровану схему DRAM у 1970 р. Intel 1103 містила 1024 біти й мала розмір слова 1 біт. Оновлення в Intel 1103 необхідно було виконувати кожні 2 мілісекунди. На початку 1970-х років пам’ять DRAM на МОН-пристроях випередила пам’ять на магнітних сердечниках як переважну технологію пам’яті в комп’ютерних системах.

DRAM є енергозалежною технологією, тобто під час вимкнення живлення заряд у конденсаторах бітових осередків вгамовується, і дані втрачаються.

Термін "**подвоєна швидкість передавання даних**" (double data rate, DDR) стосується часових характеристик обміну даними між модулем пам’яті та контролером пам’яті процесора. Попередня технологія DRAM із **базовою швидкістю передавання даних** (single data rate, SDR) передбачала виконання однієї операції передавання даних за

цикл тактового сигналу пам’яті. Пристрої пам’яті DDR виконують дві операції передавання за цикл тактового сигналу: одну – за фронтом тактового сигналу, і другу – за його спадом. Цифра після "DDR" позначає покоління технології DDR. Таким чином, DDR5 це п’яте покоління стандарту DDR. Термін "**синхронна DRAM**" (synchronous DRAM, SDRAM) вказує на те, що схема DRAM синхронізована з контролером пам’яті процесора за допомогою спільного тактового сигналу. Поточним поколінням технології DRAM, широко використовуваної в даний час, є SDRAM DDR4, а SDRAM DDR5 тільки починає виходити на ринок. Сучасні персональні комп’ютери і мобільні пристрої, такі як смартфони, як правило, містять кілька **гігабайтів (Гбайт)** оперативної пам’яті. 1 Гбайт містить 2^{30} байт, що еквівалентно 1 073 741 824 (трохи більше одного мільярда) байтів. Як випливає з назви, пам’ять довільного доступу дає змогу процесору за одну операцію зчитувати або записувати будь-яку комірку пам’яті в адресному просторі оперативної пам’яті. Але станом на 2021 р. висококласний модуль пам’яті для використання в ноутбуках містить 32 Гбайт пам’яті DRAM, розподіленої між 16 інтегральними схемами. Кожна схема DRAM у цьому модулі містить 2 гігаслова (1 гігаслово – це 2^{30} слів) при довжині одного слова 8 біт.

У 2021 р. провідним стандартом модулів пам’яті мав стати SDRAM DDR5 – вдосконалена й оптимізована технологія DRAM, що ґрунтується на поколіннях DDR1, DDR2, DDR3 і DDR4. Модуль пам’яті DDR5 поміщений у корпус типу **DIMM** (dual inline memory module – **модуль пам’яті з дворядним розташуванням виводів**). Для приєднання до роз’єму DIMM на материнській платі модуль DIMM оснащений електричними контактами по обидва боки друкованої плати (звідси термін "дворядний" у його назві). Стандартний роз’єм DIMM DDR5 має 288 контактів. Для таких систем, як ноутбуки, де вільний простір обмежений, передбачено більш компактний формат, званий **SODIMM** (small outline DIMM). Стандартний роз’єм SODIMM DDR5 має 262 контакти. Через зменшену кількість контактів модулі SODIMM позбавлені функцій, які підтримують деякі модулі DIMM, наприклад, здатності виявляти та виправляти бітові помилки в даних, які витягують із пристрою. Цю здатність забезпечує застосування **коду корекції помилок** (error correcting code, ECC).

Модулі пам’яті DDR5 зазвичай розраховані на напругу живлення 1,1 В. Показовий приклад – окремий модуль DDR5 може виконувати до 4,8 млрд операцій передачі даних за секунду, що вдвічі перевищує тактову частоту пам’яті 2400 МГц. При швидкості передачі 8 байт/с цей пристрій DDR5 теоретично може передавати за секунду 38,4 Гбайт даних. Модулі DDR5 будуть доступні з різними тактовими частотами, обсягами пам’яті та за різними цінами.

У реальних модулях DRAM використовуються прямокутні банки однобітних комірок, що були описані в попередньому розділі, однак внутрішня архітектура пристроїв DDR5 дещо складніша. Інтегральна схема DRAM зазвичай містить кілька банків.

Перед виконанням операції читання або запису логіка адресації обирає банк, що містить потрібну комірку пам'яті. У модулях DDR5 банки додатково організовано в **групи банків**, що вимагає додаткової логіки адресації для вибору потрібної групи. Пристрій DDR5 містить до 8 груп банків, кожна з яких містить до 4 банків. Причина поділу архітектури модулів DDR5 на кілька груп банків полягає в тому, щоб довести швидкість передавання даних до максимуму за рахунок паралельного виконання кількох одночасних операцій доступу до пам'яті з перекриттям. Це змогу передавати дані між процесором і оперативною пам'яттю з максимальною швидкістю, зводячи до мінімуму потребу в очікуванні завершення кожної операції доступу до DRAM.

Крім вказівки правильної адреси в модулі пам'яті DDR5, система повинна передавати за допомогою інтерфейсних сигналів команду, що вказує на дію, яку потрібно виконати: читання з пам'яті, запис у пам'ять або оновлення стану обраного рядка.

У стандарті DDR5 SDRAM, який можна придбати в **Об'єднаній раді з проектування електронних пристроїв** (Joint Electron Device Engineering Council, JEDEC) за адресою **https://www.jedec.org/standards-documents/does/jesd79-5**, наведено детальне визначення інтерфейсу пам'яті DDR5 для комп'ютерних систем. Цей стандарт містить усю інформацію, необхідну для розроблення модулів пам'яті, сумісних із будь-якою комп'ютерною системою, що підтримує DDR5.

Історично склалося так, що кожне нове покоління стандартів DDR SDRAM було несумісне з попередніми поколіннями. Материнська плата, створена для модулів пам'яті DDR5, працюватиме тільки з модулями DDR5. Роз'єм для кожного покоління модулів DDR влаштований так, щоб вставити в нього модуль іншого покоління неможливо. Наприклад, модуль DRAM DDR4 несумісний із роз'ємом DDR5.

У міру розвитку технологій пам'яті основними поліпшеннями в кожному новому поколінні є збільшення швидкості передачі даних і збільшення щільності пам'яті. Для досягнення цих цілей у більш пізніх поколіннях було зменшено напругу живлення, що дало змогу знизити енергоспоживання системи та створити щільніші схеми пам'яті, уникаючи при цьому надмірного нагрівання. Більшість сучасних процесорів сприймають системну пам'ять як лінійний масив послідовних адрес. Процесори з менш витонченою архітектурою, такі як 6502, безпосередньо звертаються до мікросхем оперативної пам'яті, використовуючи адреси пам'яті, які вказуються в інструкціях. Через складність керуючих сигналів і логіки керування банками пам'яті в пристроях SDRAM DDR5 сучасні комп'ютерні системи оснащують контролером пам'яті, який перетворює кожну лінійну адресу процесора на сигнали керування та адресації, що обирають відповідний модуль DDR5 (у системі з кількома модулями пам'яті), групу банків, банк і розташування рядків/стовпчиків у вибраному банку. Контролер пам'яті – це послідовний логічний пристрій, що керує обміном даними між процесором і модулями оперативної пам'яті DRAM. Для досягнення максимальної

продуктивності системи контролер пам’яті повинен інтелектуально використовувати можливості перекриття операцій, що надаються модулями пам’яті DDR5.

Складні сучасні процесори зазвичай інтегрують функцію контролера пам’яті в саму інтегральну схему процесора. Також можна розробити систему з окремим контролером пам’яті, який розташовується між процесором і оперативною пам’яттю.

Інтерфейс контролера пам’яті може містити кілька каналів, де кожний канал являє собою окремий шлях обміну даними між процесором і одним або декількома модулями пам’яті. Перевага виділення декількох каналів в архітектурі пам’яті полягає в тому, що така конфігурація дозволяє реалізувати одночасний доступ до пам’яті за цими каналами. Однак система, що містить кілька каналів доступу до пам’яті, не забезпечує автоматичного збільшення швидкості доступу до пам’яті. Системне програмне забезпечення повинне активно керувати призначенням областей пам’яті для кожного додатка або системного процесу, щоб збалансувати використання пам’яті між каналами. Якби операційна система просто послідовно розподіляла процеси по ділянках фізичної пам’яті, спочатку заповнюючи один модуль пам’яті, а потім переходячи до наступного, то багатоканальний доступ до пам’яті не приніс би жодної користі, оскільки всі процеси були б змушені використовувати один і той самий канал пам’яті.

DDR для графіки

DDR для графіки (graphics DDR, GDDR) – це технологія пам’яті DDR, оптимізована для використання в якості оперативної відеопам’яті в графічних контролерах. GDDR має ширшу шину пам’яті для підтримки висунутих засобами відображення вимог високої пропускної здатності. Стандартна пам’ять DDR, з іншого боку, оптимізована для забезпечення мінімальної затримки доступу до даних.

Номери поколінь технологій GDDR і DDR не узгоджені один з одним. Станом на 2021 р. модулі GDDR6 доступні вже кілька років, тоді як стандарт SDRAM DDR6, як і раніше, перебуває в розробленні.

Попередня вибірка

Одним з атрибутів продуктивності DRAM, який вельми незначно поліпшується від одного покоління до іншого, є швидкість читання або запису конкретного біта. Для того щоб домогтися збільшення середньої швидкості передавання даних у модулі DRAM і з них, пристрої повинні використовувати інші способи підвищення продуктивності. Одним із методів досягнення більш високої середньої швидкості передачі даних є **попередня вибірка**.

Ідея **попередньої вибірки** полягає у використанні того факту, що щоразу, коли процесор звертається до певного осередку пам’яті, з’являється вірогідність, що незабаром він звернеться до адрес, близьких до цього вихідного осередку. Ідея

попередньої вибірки полягає в читанні більшого блоку пам’яті, ніж одна адреса, на яку посилається інструкція процесора, і передачі всього блоку з пристрою DRAM у процесор. У контексті модуля пам’яті DDR5 розмір блоку зазвичай становить 64 байти.

Модуль DDR5 може швидко прочитати 64 байти, оскільки він звертається до всіх 512 бітів цих 64 байтів одночасно. Для цього модуль DDR5 зчитує ціле число, кратне 512 бітам бітів, з комірок, обраних шиною слів. Біти обраного ряду зчитуються одночасно, потім проходять через мультиплексор для витягання потрібних 512 біт із (можливо) 8192 біт усього ряду, які потім фіксуються у вихідному регістрі. Зафіксовані біти передаються з модуля DRAM у процесор за тактовим сигналом DDR.

За ефективного використання декількох груп банків множинні операції читання пам’яті та передавання отриманих даних можуть перекриватися в часі та забезпечувати переміщення даних між модулем пам’яті та процесором із максимальною швидкістю, яку може підтримувати інтерфейс.

Одержавши блок завдовжки 64 байти, процесор зберігає ці дані у внутрішній кеш-пам’яті та вибирає конкретний елемент даних (можливо, розміром лише 1 байт) із блоку, запитуваного виконуваною інструкцією. Якщо наступна інструкція звертається до інших даних, що містяться в тому самому блоці, процесору досить звернутися до локального кеїну, що призводить до набагато більш швидкого виконання цієї операції, ніж за допомогою інструкції, що першочергово витягувала цей блок даних з модуля DRAM.

Крім взаємодії з основною пам’яттю, процесор повинен обмінюватися даними із зовнішнім світом через пристрої введення і виведення. У наступному розділі розглядається реалізація інтерфейсів введення-виведення в сучасних комп’ютерних системах.

Підсистема введення-виведення

У *розділі 3* було представлено дві великі категорії архітектур введення-виведення: введення-виведення з розподілом пам’яті та введення-виведення з розподілом за портами. Плюси і мінуси кожного з цих підходів відігравали істотну роль на зорі розбудови персональних комп’ютерів, коли кількість фізичних адресних ліній обмежувала загальний простір пам’яті процесора діапазоном в 1 Мбайт. Архітектури сучасних процесорів можуть адресувати набагато більший діапазон пам’яті, зазвичай у десятки гігабайт. Наслідком такого розширення адресного простору пам’яті є доступність адресних областей для використання в інтерфейсі вводу-виводу. Завдяки цьому сучасні 32- і 64-розрядні процесори загального призначення використовують введення-виведення з розподілом пам’яті для більшості своїх інтерфейсів.

Складні сучасні процесори зазвичай реалізують контролер пам’яті на мікросхемі процесора, безпосередньо обмінюючись даними з модулями пам’яті DDR. Більшість

інших типів операцій введення-виведення, які виконують ці процесори, передають одній або декільком зовнішнім інтегральним схемам, які зазвичай називають **мікропроцесорним набором** або **чіпсетом**. Термін "чіпсет", як правило, використовується навіть у тих випадках, коли для реалізації функцій вводу-виводу потрібна всього одна мікросхема.

Чіпсет надає інтерфейси для широкого спектра периферійних пристроїв, включно з дисковими накопичувачами, мережевими адаптерами, клавіатурами, мишами і багатьма іншими пристроями з підтримкою USB. Більшість цих інтерфейсів реалізовано на основі того чи іншого виду послідовної шини. У наступних розділах представлено найпоширеніші технології введення-виведення, що використовуються в сучасних комп’ютерах.

Паралельні та послідовні шини даних

Паралельна шина даних одночасно передає кілька бітів даних окремими провідниками між двома або великим числом кінцевих точок. У ранніх ПК паралельні шини використовувалися для таких функцій, як підключення принтера до комп’ютера. Згодом стали очевидні деякі обмеження паралельних шин.

- Залежно від розрядності для підключення паралельної шини може знадобитися багато проводів. Це означає, що кабелі коштують дорожче і існує підвищена ймовірність виникнення проблем через обрив дротів кабелю або ненадійного електричного контакту в роз’ємах.
- У міру того, як розробники комп’ютерних систем намагалися збільшити швидкість передавання даних шиною (і тим самим одержати конкурентну перевагу), істотну роль стало відігравати ще одне обмеження паралельних шин: незважаючи на те, що пристрій, який передає слово даних шиною, здатний виводити всі його біти одночасно, окремі сигнали можуть прибувати до точки призначення в різний час.
- Це може бути викликано відмінностями в ефективній довжині шляху в провідниках кабелю або друкованої плати. Цією обставиною обумовлено існування верхньої межі швидкості передачі даних, яку може підтримувати архітектура паралельної шини.

Ще одне обмеження паралельних шин – вони можуть передавати дані одночасно тільки в одному напрямку (так званий **напівдуплексний режим**), якщо не передбачено дублюючого набору з’єднань для одночасного передавання даних у протилежному напрямку. З цієї причини паралельні шини зазвичай не забезпечують одночасного двостороннього зв’язку, який називають **повнодуплексним режимом**.

Послідовна шина даних передає дані між двома кінцевими точками по одному біту за раз за допомогою пари провідників. Більшість високошвидкісних каналів зв’язку між

процесором і периферійними пристроями в сучасних комп'ютерах використовують той чи інший варіант послідовної шини. На перший погляд, перехід від паралельної шини до послідовної тягне за собою істотну втрату пропускної здатності, однак послідовні шини мають низку важливих переваг, які роблять їхнє застосування привабливим під час розв'язування завдань, які висувають високі вимоги до продуктивності.

Високошвидкісні послідовні шини в персональних і комерційних комп'ютерах передають дані парами провідників з використанням **диференціальної передачі сигналів**. У цьому випадку використовуються два провідники, які ретельно підібрані за довжиною і мають майже однакові електричні характеристики. У кабелях ці провідники являють собою ізольовані дроти, скручені один з одним для утворення кручених пар.

На рис. 4.7 представлено послідовну шину даних з диференціальною передачею сигналів.

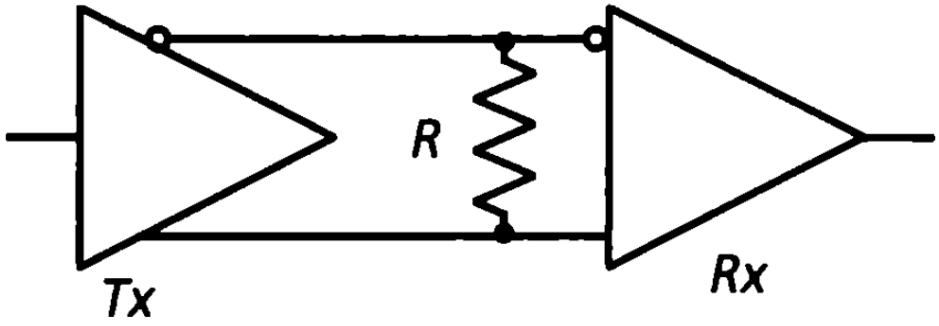


Рис 4.7. *Схема послідовної шини з диференціальною передачею сигналів*

Цифровий сигнал, який необхідно передати, надходить у передавач (позначений **Tx**) через вхід у лівій частині малюнка. Вхідний сигнал перетворюється на напругу на двох паралельних лініях, показаних у середній частині схеми. Маленький кружок вказує на те, що сигнал, який іде від передавача верхньою лінією, інвертований відносно сигналу в нижній лінії.

У типовому послідовному інтерфейсі високий рівень сигналу на вході передавача генеруватиме напругу 1,0 В на верхньому провіднику послідовної шини і 1,4 В на нижньому провіднику. Вхідний сигнал низького рівня призведе до появи напруги 1,4 В на верхньому провіднику і 1,0 В на нижньому провіднику.

Входи приймача (позначеного **Rx**) мають високий опір – це означає, що приймач споживає незначний струм із ланцюга. Приймач вимірює напругу на резисторі, типовий опір якого становить 100 Ом. Коли на вході **Tx** високий рівень, верхній вивід резистора перебуває під напругою Ю,4 В відносно нижнього виводу. Коли на вході **Tx** низький рівень, верхній вивід резистора перебуває під напругою +0,4 В відносно нижнього виводу.

Приймач генерує свій вихідний сигнал, інвертуючи один із входів (верхній на рис. 4.7, з маленьким кружком) і додаючи отриману напругу до напруги на іншому вході. Іншими словами, приймач вимірює тільки різницю між напругами на двох провідниках. Фундаментальна перевага цього підходу полягає в тому, що більшість перешкод, які спотворюють сигнали, спричиняють коливання напруги в провідниках, що несуть сигнал. При розміщенні двох провідників дуже близько один до одного більша частина шумової напруги, що вноситься в один із провідників, також буде проявлятися і в іншому провіднику. Операція віднімання усуває більшу частину шуму, який в іншому разі заважав би точному виявленню сигналу приймачем.

Послідовна шина даних може передавати кілька мільярдів бітів на секунду, що набагато більше, ніж паралельна шина в старих ПК. Крім того, можна використовувати кілька послідовних шин поруч одна з одною, що дає змогу помножити пропускну здатність залежно від кількості шин.

Принципова відмінність між кількома послідовними шинами, що з'єднують дві кінцеві точки, і паралельною шиною, що реалізує таке ж з'єднання, полягає в тому, що в багатьох стандартах інтерфейсів послідовні шини працюють до певної міри незалежно. Для них не потрібна синхронізація на рівні передавання кожного біта, як у разі передавання паралельною шиною. Це спрощує влаштування з'єднань, здатних підтримувати дуже високі швидкості передавання даних, водночас потрібно піклуватися лише про точну відповідність довжини провідника та електричних характеристик у кожній парі послідовних провідників.

Зв'язок між сучасним процесором і чіпсетом його материнської плати зазвичай здійснюється за кількома послідовними лініями даних, які називаються **лініями високошвидкісного введення-виведення** (high-speed input output, HSIO). Кожна лінія являє собою послідовне з'єднання з одним каналом передавання даних у кожному напрямку (наприклад, таким, як на рис. 4.7), завдяки чому обмін даними здійснюється в повнодуплексному режимі.

Окремі лінії HSIO можуть бути призначені певним типам периферійних інтерфейсів, реалізованих у вигляді послідовних з'єднань, таких як PCI Express, SATA, M.2, USB і Thunderbolt. У наступних розділах представлено кожен із цих стандартів.

PCI Express

Оригінальна шина Peripheral Component Interconcept (PCI) була 32-розрядною паралельною шиною, що працювала на частоті 33 МГц, яка використовувалася в IBM PC-сумісних комп'ютерах приблизно з 1995 до 2005 р. У роз'єми PCI на материнських платах комп'ютерів можна було встановити безліч карт розширення з різними функціями, включно з мережевим інтерфейсом, адаптером дисплея або платою звукового виходу. До початку 2000-х років обмеження архітектури паралельної шини стали стримувати подальший розвиток, і почалася розробка заміни PCI у вигляді

послідовної шини під назвою PCI Express. PCI Express (PCIe) являє собою двонаправлену послідовну шину з диференціальним передаванням сигналів, використовувану переважно для з'єднання кінцевих точок обміну даними на материнських платах комп'ютерів. Пропускна здатність PCIe виражається в мільярдах пересилань даних (гігатранзакцій) за секунду або ГТ/с. Одна транзакція – це передання одного біта по шині від передавача до приймача. У кожен багато-бітний пакет PCIe додає додаткові надлишкові біти для контролю цілісності даних. Різні покоління PCIe використовують різну кількість цих службових бітів, що впливають на фактичну швидкість передачі даних. У табл. 4.1 наведено дані основних поколінь PCIe із зазначенням року їхньої появи, швидкості передавання однією лінією в гігатранзакціях за секунду (ГТ/с) і ефективної швидкості передавання даних у мегабітах за секунду (Мбіт/с).

Таблиця 4.1. *Покоління шини PCI Express*

Покоління PCIe	Рік випуску специфікації	Швидкість передачі, ГТ/с	Ефективна швидкість передачі даних в одну сторону, Мбіт/с
1.0a	2003	2,5	250
2.0	2007	5	500
3.0	2010	8	985
4.0	2017	16	1969
5.0	2019	32	3938
6.0 (запропоновано)	2021	64	7877

Наведена тут ефективна швидкість передавання даних вказана для одностороннього передавання даних. PCIe підтримує передавання даних на повній швидкості одночасно в обох напрямках.

Стандарти PCIe підтримують з'єднання за кількома лініями, число яких позначається як x1, x2, x4, x8, xl6 і x32. Більшість сучасних материнських плат оснащуються як мінімум роз'ємами PCIe xl і xl6. Роз'єми PCI x1 сумісні з крайовим роз'ємом плати завдовжки 25 мм, а для роз'ємів x16 потрібен крайовий роз'єм завдовжки 89 мм. Карта PCIe буде коректно працювати в будь-якому роз'ємі, в який вона фізично поміщається. Наприклад, карту PCIe x1 можна вставити в роз'єм xl6, і вона буде використовувати тільки одну з 16 доступних ліній.

Основна сфера застосування роз'ємів PCIe x16 – інтерфейс між процесором і графічною картою для забезпечення максимальної продуктивності для застосунків з інтенсивним використанням графіки, як-от ігри. Інтерфейс PCIe 5.0 xl6 забезпечує одностороннє передавання даних зі швидкістю 63 Гбіт/с.

У сучасних комп’ютерних архітектурах чіп процесора зазвичай забезпечує пряме з’єднання з графічною картою, встановленою в роз’єм PCIe xl6, через 16 ліній PCIe. Це означає, що передавати сигнали графічної карти PCIe через чипсет не потрібно.

Крім інтерфейсів графічної карти і DDR SDRAM, керування більшою частиною операцій введення-виведення в сучасних комп’ютерних системах здійснює чіпсет. Процесор і чіпсет взаємодіють через набір ліній HSIO. Чіпсет забезпечує інтерфейси для периферійних пристроїв, таких як дискові накопичувачі, мережеві інтерфейси, клавіатура і миша. Інтерфейси для цих пристроїв зазвичай використовують стандарти послідовного інтерфейсу SATA, M.2 і USB, які обговорюються далі.

SATA

Serial AT Attachment (SATA) – це стандарт двостороннього послідовного інтерфейсу для під’єднання материнських плат комп’ютерів до пристроїв зберігання даних. "AT" в абревіатурі SATA відноситься до IBM PC AT. Як і одна лінія PCIe, інтерфейс SATA містить дві пари провідників із диференціальним передаванням сигналів, де по одній парі передають дані в кожному напрямку. На відміну від PCIe, інтерфейс SATA призначений для роботи по кабелях, а не по металевих сигнальних доріжках на материнських платах. На додаток до вимог щодо електричної частини та формату даних стандарт SATA містить докладні специфікації сумісних кабелів і роз’ємів.

Кабель SATA містить одну двосторонню лінію, що забезпечує зв’язок між процесором і пристроєм зберігання, таким як накопичувач на магнітних дисках, дисковод оптичних дисків або твердотільний накопичувач. У табл. 4.2 наведено основні версії стандарту SATA із зазначенням року появи та робочих параметрів кожної з них.

Таблиця 4.2. *Покоління стандарту SATA*

Покоління SATA	Рік випуску специфікації	Швидкість передачі, ГТ/с	Ефективна швидкість передачі даних в одну сторону, Мбіт/с
1.0	2003	1,5	150
2.0	2004	3	300
3.0	2009	6	600

Швидкість передавання даних у цій таблиці наведено для одностороннього передавання, хоча, як і PCIe, SATA підтримує повно-дуплексне передавання даних. Удосконалення стандарту SATA триває, але станом на 2021 р. оголошень про наступне покоління SATA з вищою швидкістю передавання даних не було.

M.2

У сучасних **твердотільних накопичувачах** (solid-state drives, SSD) для зберігання даних використовують флеш-пам’ять, а не жорсткі магнітні диски, що обертаються, як у традиційних дискових накопичувачах. Через радикальні відмінності технології твердотільних накопичувачів інтерфейс SATA, який у більшості випадків працює досить добре з накопичувачами на обертових дисках, виявився значною перешкодою для ефективної роботи твердотільних накопичувачів.

Для того щоб отримати доступ до довільного блоку даних на магнітному диску (так званого **сектору**), голівка накопичувача повинна фізично переміститися на доріжку, яка містить цей сектор, потім дочекатися, доки початок сектора досягне положення голівки, після чого накопичувач зможе почати читання даних. Навпаки, твердотільний накопичувач може безпосередньо звертатися до будь-якого сектора даних, використовуючи спосіб, дуже схожий на доступ процесора до комірки динамічної пам’яті DRAM.

Специфікація M.2 була розроблена для створення компактного і високопродуктивного інтерфейсу доступу до флеш-пам’яті в невеликих портативних пристроях. Завдяки цьому обмеження інтерфейсу SATA було усунуто, і швидкість передавання даних збільшилася в кілька разів порівняно зі швидкістю інтерфейсу SATA.

Крім роботи із запам’ятовуючими пристроями M.2 підтримує й інші інтерфейси, зокрема PCIe, USB, Bluetooth і Wi-Fi. Сучасні материнські плати містять роз’єми M.2, які, крім більш високої пропускної здатності, займають набагато менше місця в корпусі комп’ютера, ніж традиційні відсіки для дискових накопичувачів.

USB

Інтерфейс **USB (Universal Serial Bus)** являє собою простий (з точки зору користувача) інтерфейс для підключення до комп’ютерної системи найрізноманітніших периферійних пристроїв. На додаток до протоколів зв’язку версії стандарту USB визначають вимоги до кабелів, роз’ємів і до подачі живлення на під’єднані по кабелю USB пристрої.

Кабелі USB мають легко визначувані типи роз’ємів, а пристрої з підтримкою USB підтримують "гаряче" під’єднання (під’єднання пристроїв один до одного за увімкненого живлення). USB-пристрої підтримують автоматичне налаштування під час під’єднання, і в більшості випадків користувачам не потрібно турбуватися про встановлення драйверів під час під’єднання нового пристрою до комп’ютера за допомогою USB-кабелю.

Кабелі USB ранніх версій (до покоління 2.0) містили одну пару дротів із диференціальною передачею сигналів, яка одноразово могла передавати дані тільки в одному напрямку. Пізніші версії стандарту USB (починаючи з USB 3.2 покоління 1)

підтримують одночасну двосторонню передачу даних. Крім того, USB версії 3.2 і USB4 містять до двох ліній, що дало змогу подвоїти швидкість передавання даних.

У табл. 4.3 наведено основні версії стандарту USB із зазначенням дати появи, максимальної кількості підтримуваних ліній і максимальної швидкості передачі даних для кожної з них.

Таблиця 4.3. *Покоління стандарту USB*

Покоління USB	Рік випуску специфікації	Кількість ліній	Швидкість передачі, ГТ/с	Ефективна швидкість передачі даних в одну сторону, Мбіт/с
1.1	1998	1	0,012	1,5
2.0	2000	1	0,48	60
3.2, покоління 1	2008	1	5	500
3.2, покоління 2	2013	1	10	1200
3.2, покоління 2x2	2017	2	20	2400
USB4	2019	2	40	4800

У поколіннях USB до 2.0 обмін даними повністю перебуває під керуванням комп’ютера, що керує (хоста). Хост ініціює кожну транзакцію обміну даними, відправляючи пакети, адресовані конкретному пристрою, і здійснює передачу даних у пристрій або з нього. Починаючи з USB 3.2 (покоління 1), пристрої можуть ініціювати зв’язок із хостом, що дає під’єднаним периферійним пристроям можливість генерувати переривання.

Thunderbolt

Thunderbolt – це набір стандартів високошвидкісних послідовних інтерфейсів, представлений у 2011 р. Оригінальний інтерфейс Thunderbolt поєднував передачу сигналів PCIe і DisplayPort з використанням двох послідовних ліній Thunderbolt.

Thunderbolt 4 – це останнє покоління стандарту Thunderbolt, що додає сумісність із USB4 і підтримує під’єднання пристроїв PCIe і кількох моніторів високої роздільної здатності через один комп’ютерний порт. Thunderbolt 4 використовує той самий роз’єм, що і USB 3.2 і пізніші покоління (роз’єм USB-C), і підтримує швидкість передавання даних USB4 – 40 Гбіт/с. USB-пристрої при підключенні до порту Thunderbolt 4 працюють коректно. Однак периферійні пристрої Thunderbolt 4 несумісні з портами USB-C, відмінними від Thunderbolt 4.

У наступному розділі подано огляд найпопулярніших стандартів інтерфейсів графічних дисплеїв.

Графічні дисплеї

В іграх, редакторах відеоматеріалів, графічному дизайні та анімації продуктивність обробки відеоданих має вирішальне значення. Створення та відображення графіки з високою роздільною здатністю вимагає величезного обсягу математичних обчислень.

Процесори загального призначення можуть виконувати необхідні для цього обчислення, однак їм не вистачає продуктивності, яку очікують користувачі цих додатків.

Високопродуктивні графічні карти, звані **графічними процесорами** (graphics processing units, GPU), по суті є мініатюрними суперкомп'ютерами, значною мірою оптимізованими для виконання графічних обчислювальних завдань, таких як візуалізація тривимірних сцен. Оскільки обчислення, пов'язані з візуалізацією сцен, часто повторюються, за допомогою апаратного розпаралелювання можна домогтися істотного підвищення продуктивності. Графічні процесори містять велику кількість відносно простих обчислювальних блоків, кожен з яких виконує невелику частину загального завдання.

Графічний процесор може містити тисячі окремих процесорів, кожен з яких виконує функції АЛУ. Початковою рушійною силою, що призвела до розроблення високопродуктивних графічних процесорів, була побудова тривимірних сцен, однак пізніші покоління цієї технології знайшли широке застосування в таких галузях, як аналіз великих даних і машинне навчання. Архітектура графічного процесора дає змогу прискорити розв'язання будь-якої обчислювальної задачі з інтенсивними обчисленнями, яку можна розбити на набір паралельних операцій.

Звичайно, не всім користувачам потрібна надвисока продуктивність під час обробки відеоданих. Для задоволення потреб користувачів зі скромними вимогами до роботи з графікою й обмеженим бюджетом у сучасні процесори часто вбудовують графічний процесор середньої продуктивності. У багатьох додатках такий підхід забезпечує більш ніж достатню продуктивність обробки графічних даних. Така конфігурація називається **конфігурацією з інтегрованою графікою** – це означає, що функція графічного процесора інтегрована в кристал процесора та використовує загальну системну пам'ять спільно з процесором. Комп'ютерні системи з інтегрованою графікою дешевші, але забезпечують достатню графічну продуктивність для розв'язання основних обчислювальних завдань, таких як робота з електронною поштою, перегляд веб-сторінок і відеозаписів.

Багато настільних комп'ютерних систем, а також деякі ноутбуки мають вбудований графічний процесор, пропонуючи можливість встановлення високопродуктивної графічної карти. Це дає користувачам можливість адаптувати комп'ютерну систему до своїх потреб за ціною та продуктивністю.

Нині для під’єднання моніторів до комп’ютерів використовується кілька різних стандартів відеосигналів. Оскільки вихідні сигнали, що генеруються графічним інтерфейсом комп’ютера, мають бути сумісні з під’єднаним монітором, комп’ютери зазвичай оснащують відеороз’ємами кількох типів. Комп’ютерні монітори і телевізори високої чіткості зазвичай також пропонують вибір інтерфейсів підключення.

У розділі 6 архітектура засобів обробки графічних процесорів буде розглянута більш докладно. У наступних розділах описуються стандарти комп’ютерних відеоінтерфейсів минулого і сьогодення.

VGA

Відеостандарт **Video Graphics Array (VGA)** для персональних комп’ютерів був представлений компанією IBM у 1987 р. VGA – це аналоговий інтерфейс, який широко застосовується і сьогодні, хоча більшість сучасних комп’ютерів не мають роз’єму VGA. Нерідко старі комп’ютери з відеовиходами VGA використовують кабель-перехідник для передавання зображення на монітор, що підтримує відеовхід DVI або HDMI.

Сучасні версії стандарту VGA підтримують роздільну здатність до 1920 пікселів завширшки і 1200 пікселів заввишки з частотою оновлення 60 Гц. Оскільки відеосигнал VGA є аналоговим, під час передавання на монітор відбувається деяка втрата якості сигналу. Цей ефект найпомітніший за високої роздільної здатності екрана.

DVI

Відеостандарт **Digital Visual Interface (DVI)** було розроблено для поліпшення візуальної якості комп’ютерного зображення завдяки передачі з комп’ютера на монітор цифрового відеосигналу. Для забезпечення зворотної сумісності зі старими комп’ютерами і моніторами по кабелях DVI також можна передавати аналогові сигнали VGA.

Подібно до високошвидкісних послідовних інтерфейсів, розглянутих раніше в цьому розділі, для передавання відеоданих DVI використовує послідовне диференціальне передавання сигналів. Роз’єм DVI містить чотири послідовні лінії. Окремі лінії несуть інформацію про червоний, зелений і синій кольори, а по четвертій лінії передається загальний тактовий сигнал.

Визначено три варіанти DVI залежно від комбінації підтримуваних типів цифрових і аналогових відеосигналів.

- **DVI-A** підтримує тільки аналоговий відеосигнал. Цей варіант інтерфейсу призначений для забезпечення зворотної сумісності з комп’ютерами та моніторами VGA. Роз’єм DVI-A має інше розташування контактів порівняно з

традиційними роз'ємами VGA, тому для під'єднання до застарілих пристроїв VGA потрібен кабель-перехідник.

- **DVI-D** являє собою винятково цифровий інтерфейс, що підтримує одноканальний і двоканальний варіанти. Двоканальний варіант надає додаткові лінії послідовного передавання даних, щоб збільшити пропускну здатність для дисплеїв із вищою роздільною здатністю. "Двоканальний" не означає, що кабель підтримує два монітори.
- **DVI-I** являє собою інтегрований інтерфейс, що підтримує як аналоговий інтерфейс DVI-A, так і цифрові режими DVI-D. Цифровий інтерфейс DVI-I може бути як одноканальним, так і двоканальним.

Інтерфейси DVI використовують переважно в комп'ютерних моніторах. Ефективна швидкість передачі даних одноканального з'єднання DVI-D становить 3,96 Гбіт/с. Двоканальний DVI-D передає відеодані вдвічі швидше, ніж одноканальний: 7,92 Гбіт/с.

HDMI

Мультимедійний інтерфейс високої чіткості **High-Definition Media Interface (HDMI)** підтримується більшістю сучасних комп'ютерів і моніторів, а також практично всіма сучасними телевізорами і пов'язаними з ними розважальними відеопристроями, такими як DVD-плеєри. HDMI підтримує тільки цифрові відеодані (підтримка аналогових сигналів відсутня) і використовує ту саму диференціальну послідовну шину, що і DVI-D. На додаток до відеоданих кабелі HDMI також передають багатоканальний цифровий аудіосигнал.

З моменту введення в 2002 р. стандарт HDMI зазнав кілька редакцій. Кожна наступна версія підтримувала зворотну сумісність, додаючи нові можливості. У пізніх версіях стандарту збільшено пропускну спроможність, розширено діапазон підтримуваних роздільних здатностей екрана, додано підтримку звуку високої чіткості та Ethernet підключення по кабелю HDMI, а також функції для підтримки ігор. Кожна версія HDMI сумісна з попередніми версіями, однак нові функції доступні тільки в конфігураціях, у яких пристрій-джерело сигналу, пристрій відображення і з'єднувальний кабель сумісні з новим стандартом.

Стандарт HDMI версії 2.1 був опублікований у 2017 р. Він підтримує ефективну швидкість передавання даних 42,6 Гбіт/с за чотирма диференціальними послідовними лініями.

DisplayPort

Стандарт DisplayPort, введений у 2006 р., описує цифровий інтерфейс, що підтримує цифрове відео, аудіо та USB-з'єднання. Тоді як стандарт HDMI орієнтований на

побутову електроніку, включно з телевізорами і домашніми кінотеатрами, DisplayPort націлений на обчислювальну техніку. DisplayPort передає дані в пакетах із вбудованою в кожен пакет інформацією про синхронізацію, що усуває необхідність в окремому каналі синхронізації.

Один вихід DisplayPort комп’ютера може керувати кількома моніторами, сполученими в гірляндний ланцюг, де один кабель з’єднує комп’ютер із першим монітором, другий кабель з’єднує перший і другий монітори тощо. Монітори, що використовуються, повинні підтримувати таку функцію. Максимальна кількість моніторів, які можна з’єднати в такий спосіб, обмежена тільки можливостями відеокарти, максимальною пропускною спроможністю кабелю, а також роздільною здатністю і частотою оновлення моніторів.

Версія DisplayPort 2.0, випущена 2019 року, підтримує ефективну швидкість передавання даних до 77,4 Гбіт/с за чотирма диференціальними послідовними лініями.

Мережевий інтерфейс

Комп’ютерна мережа – це сукупність цифрових пристроїв, що взаємодіють через загальне середовище зв’язку. **Локальна мережа** (local area network, LAN) складається з обмеженої кількості комп’ютерів, які можуть перебувати в одному фізичному місці, наприклад у житловому будинку або в офісній будівлі. Підключені один до одного комп’ютери, телефони та інші цифрові пристрої у вашому будинку являють собою локальну мережу. Для з’єднання пристроїв у середовищі локальної мережі можна використовувати дротовий інтерфейс, зазвичай Ethernet, або бездротовий інтерфейс, зазвичай Wi-Fi.

Територіально віддалені один від одного комп’ютери та локальні мережі обмінюються даними за допомогою **глобальної мережі** (wide area network, WAN). Сервіси глобальної мережі часто надаються операторами зв’язку, такими як постачальники кабельного телебачення або телефонні компанії. Ваша домашня локальна мережа, найімовірніше, під’єднана до Інтернету через сервіс глобальної мережі, який надає ваш оператор телефонного зв’язку або кабельної мережі.

Пристрої мережевого інтерфейсу для домашніх і корпоративних мереж (маршрутизаторів), що надаються постачальниками послуг глобальної мережі, зазвичай пропонують підтримку Ethernet або Wi-Fi для під’єднання локальних пристроїв до глобальної мережі. У наступних розділах представлені технології Ethernet і Wi-Fi.

Ethernet

Ethernet являє собою набір мережевих стандартів для з’єднання комп’ютерів у локальну мережу за допомогою кабелів. Початкова версія Ethernet була розроблена в

1974 р. Робертом Меткалфом (Robert Metcalfe), який працював у дослідницькому центрі Xerox у Пало-Альто. Стандарт Ethernet було запущено в комерційну експлуатацію в 1980 р. як технологію зв'язку зі швидкістю 10 Мбіт/с для груп комп'ютерів, з'єднаних коаксіальним кабелем. Назва технології походить від історичного терміну luminiferous aether (світлоносний ефір), який означає гіпотетичне середовище, яке заповнює весь простір і забезпечує поширення електромагнітних хвиль. Кабель Ethernet слугує концептуально схожим середовищем зв'язку.

Інститут інженерів з електротехніки та електроніки (Institute of Electrical and Electronic Engineers, IEEE) розпочав розробку стандартів для технологій локальних мереж, включно з Ethernet, у 1980 р. Стандарт Ethernet IEEE 802.3 було опубліковано в 1985 р. Відтоді він зазнав низки змін, зокрема збільшення швидкості передавання даних і додавання різних топологій мережі. Найочевиднішою відмінністю сучасних комп'ютерних мереж від вихідного стандарту Ethernet є використання двожильних кабелів із крученими парами замість загального коаксіального кабелю.

Сучасні комп'ютери зазвичай використовують інтерфейси Gigabit Ethernet для обміну даними кабелями з **неекранованими крученими парами** (unshielded twistedpair, UTP). Інтерфейс Gigabit Ethernet офіційно визначено в стандарті IEEE 802.3ab, він підтримує швидкість передавання даних 1,0 Гбіт/с (ефективна швидкість досягає 99% вихідної швидкості), хоча кількість доданих службових даних значно варіює залежно від використовуваного протоколу зв'язку.

Передавання даних через інтерфейс Ethernet здійснюється блоками даних змінного розміру, що називаються **кадрами**, які можуть містити до 1518 байт. Заголовок кожного кадру містить адресну інформацію, що ідентифікує інтерфейси Ethernet джерела та одержувача. Оскільки сучасні з'єднання по кручений парі є двожильними, найпоширенішою схемою для з'єднання групи комп'ютерів є прокладання кабелю від кожного комп'ютера до комутатора. Комутатор – це пристрій, який приймає кадри, що передаються під'єднаними комп'ютерами, і на основі адреси одержувача, зазначеної в кожному кадрі, відразу ж пересилає їх правильному одержувачу. Максимальна рекомендована довжина кабелів Ethernet становить 100 метрів, що обмежує фізичний розмір локальної мережі Ethernet такою областю, як окрема офісна будівля або житловий будинок.

Сучасні материнські плати зазвичай містять вбудований інтерфейс Ethernet, що позбавляє від необхідності використовувати роз'єм PCIe з картою Ethernet. Інтерфейс Ethernet, вбудований у материнську плату або встановлений у роз'єм розширення PCIe, використовує одну лінію HSIO, яка з'єднує процесор через чіпсет з інтерфейсом Ethernet.

Wi-Fi

Інститут IEEE випустив першу версію стандарту бездротового зв’язку 802.11 у 1997 р. Ця версія підтримувала швидкість передавання даних 2 Мбіт/с у частотному діапазоні 2,4 ГГц. Стандарт 802.11b, випущений у 1999 р., який підтримував швидкість передавання даних 11 Мбіт/с, виявився комерційно успішним. Ця технологія отримала назву Wi-Fi у 1999 р. з відсиланням до терміне "hi-fi", що означає високоточне відтворення звуку.

Версія 802.11g, випущена в 2003 р., підтримує вихідну швидкість передачі даних 54 Мбіт/с. Версія 802.11n, випущена в 2009 р., також підтримує **технологію використання декількох передавальних і декількох приймальних антен** (multiple-input-multiple-output, MIMO) і роботу в діапазоні 5 ГГц. Стандарт 802.11ac, опублікований у 2013 р., підтримує швидкість передавання даних у діапазоні 5 ГГц понад 500 Мбіт/с у разі використання конфігурацій з антенами MIMO. Стандарт 802.11ax, затверджений у 2021 р., забезпечує збільшення пропускної спроможності на 39%, порывнюючи зі стандартом 802.11ac для одного клієнта, і пропонує розширену підтримку під час використання в середовищах із високою щільністю радіосигналів. Версія 802.11ax просувається під найменуванням "Wi-Fi 6" для роботи в діапазонах 2,4 і 5 ГГц і "Wi-Fi 6E" – для роботи в діапазоні 6 ГГц.

Пристрої Wi-Fi можуть страждати від перешкод, створюваних побутовими приладами, такими як бездротові телефони, мікрохвильові печі та інші мережі Wi-Fi, що працюють поблизу. На поширення сигналу Wi-Fi впливають такі чинники, як стіни та інші перешкоди між передавачем і приймачем, **багатопроменеве поширення сигналу** (послаблююча інтерференція між сигналом, що поширюється прямим шляхом, і його відбитою копією), а також максимальна потужність, яку дозволено випромінювати передавачам Wi-Fi. Використання декількох антен у конфігураціях 802.11n, 802.1 lac і 802.11 ax значно знижує вплив багатопроменевого поширення сигналів на якість передавання даних.

Сучасні маршрутизатори глобальних мереж, що надаються операторами зв’язку, зазвичай містять комбінацію інтерфейсів Ethernet і Wi-Fi. Основною перевагою Wi-Fi порівняно з Ethernet для цих додатків є скорочення кількості використовуваних кабелів.

Одним із недоліків Wi-Fi є можливе порушення безпеки, оскільки радіочастотний сигнал може поширюватися далеко за межі будівлі, в якій знаходяться системи зв’язку. Стандарти Wi-Fi забезпечують необхідну підтримку безпеки зв’язку за допомогою таких протоколів, як **Wi-Fi Protected Access 2 (WPA2)**, але системні адміністратори та користувачі повинні переконатися в тому, що відповідні функції безпеки ввімкнено, а секретні дані, як-от пароль доступу до мережі, який при цьому має бути достатньо складним, надійно захищені.

Підтримка Wi-Fi реалізована в більшості портативних мобільних пристроїв, таких як ноутбуки, смартфони та планшети, і безпосередньо вбудована в багато материнських плат.

У наступному розділі представлені комп'ютерні інтерфейси з найменшими вимогами до пропускної спроможності: клавіатура і миша.

Клавіатура і миша

Порівняно з високошвидкісними інтерфейсами, розглянутими раніше в цій главі, вимоги до пропускної спроможності для клавіатури і миші досить скромні. Ці пристрої підтримують єдині методи введення, використовувані людиною-оператором у більшості конфігурацій комп'ютерів, тому від них вимагається робота зі швидкістю людських дій. Навіть найшвидша друкарка може натискати клавіші зі швидкістю не більш як один-два десятки натискань на секунду.

Клавіатура

Механічна комп'ютерна клавіатура складається з набору клавіш, кожна з яких приводить у дію електричний перемикач миттєвої дії. Стандартна повно розмірна клавіатура зазвичай містить 104 клавіші, включно з клавішами зі стрілками, клавішами керування (<Home>, <Scroll Lock>, тощо) і додатковою клавіатурою для введення цифр. Для під'єднання до комп'ютера сучасні клавіатури зазвичай використовують USB-кабель або бездротовий зв'язок.

Оскільки вимоги до пропускної здатності під час взаємодії з людиною настільки малі, деякі материнські плати комп'ютерів оснащені відносно повільним портом USB 2.0 для під'єднання клавіатури, водночас пропонуючи швидші інтерфейси USB 3.2 і вище для високошвидкісних периферійних пристроїв. Це призводить до невеликого зниження вартості компонентів материнської плати.

Оскільки клавіатура повідомляє про натискання і відпускання кожної клавіші окремо, комп'ютер може обробляти одночасні натискання комбінацій клавіш. Наприклад, результатом натискання клавіші <A> при утриманні натиснутою клавіші <Shift> є введення великої літери A.

Деякі комп'ютери та мобільні пристрої, такі як планшети та телефони, оснащені інтерфейсом із сенсорним екраном. Коли на цих пристроях потрібно ввести текст, система відображає на екрані клавіатуру, і користувач торкається літер, щоб натиснути клавіші.

Механічні клавіатури, як правило, забезпечують більш точне введення, і їх віддають перевагу користувачі, які вводять значну кількість тексту. Оскільки поверхня сенсорного екрану абсолютно плоска, пальці користувача не відчувають натискання клавіш. Це призводить до більш частих помилок введення при використанні

клавіатури на сенсорному екрані. І звичайно ж, при наявності клавіатури, що відображається на сенсорному екрані, немає необхідності використовувати механічну клавіатуру, що є істотною перевагою для мобільних пристроїв. Крім того, введення за допомогою сенсорної клавіатури не схильне до фізичних відмов, які можуть виникати в механічних деталях клавіатур. Для користувачів в рукавичках введення тексту ускладнене як на сенсорних екранах, так і на клавіатурах.

Миша

Комп’ютерна миша — це ручний пристрій, який переміщує покажчик по екрану комп’ютера в горизонтальному і вертикальному напрямках. Користувач ініціює необхідні дії в залежності від місця розташування покажчика, натискаючи кнопки миші. Сучасні миші часто мають невелике коліщатко, здатне обертатися в будь-якому напрямку і використовується для виконання таких завдань, як прокрутка документа.

Як і клавіатура, миша зазвичай підключається до комп’ютера через інтерфейс USB по дротовому або бездротовому з’єднанню. Миша пред’являє низькі вимоги до пропускної здатності, для задоволення яких достатньо порту USB 2.0.

Для роботи миші потрібна рівна горизонтальна поверхня, зазвичай стільниця, по якій користувач переміщує мишу. Сучасні миші найчастіше використовують оптичні випромінювачі і датчики для виявлення їх переміщення по поверхні. Багатьом моделям мишей важко працювати на сильно відбивних поверхнях, таких як скляні стільниці.

Трекбол за своєю концепцією схожий на мишу, за винятком того, що замість переміщення миші по поверхні кулька утримується у фіксованому положенні, але її можна обертати в будь-якому напрямку за допомогою рухів руки. Катаючи кульку вперед, назад, вліво і вправо, користувач може переміщати покажчик на дисплеї.

Трекбол не вимагає стільки місця на поверхні, скільки потрібно миші, і його можна встановити стаціонарно. Можливість жорстко закріпити трекбол робить його кращим вказівним пристроєм для комп’ютерних станцій, встановлених на транспортних засобах, включаючи наземний, морський і повітряний транспорт.

Як і у випадку з клавіатурою, комп’ютер визначає натискання і відпускання кожної кнопки миші як окремі події. Користувачі можуть використовувати цю можливість для виконання таких операцій, як перетягування значка по екрану. Для цього потрібно виконати наступні дії:

1. Наведіть курсор на значок.
2. Натисніть і утримуйте ліву кнопку миші.
3. Перемістіть курсор (з прикріпленим до нього значком) у потрібне місце.
4. Відпустіть кнопку миші.

Разом клавіатура і миша забезпечують всі можливості введення, необхідні більшості користувачів комп'ютерів для виконання інтерактивних завдань.

У наступному розділі зібрані описи підсистем з цього розділу для оцінки характеристик материнської плати сучасного комп'ютера.

Технічні характеристики сучасної комп'ютерної системи

За допомогою інформації, що міститься в цьому розділі, ви зможете розібратися в більшості технічних характеристик материнської плати, процесора і чіпсета сучасного комп'ютера. У цьому розділі наведено приклад характеристик сучасної (на 2021 р.) материнської плати з деякими поясненнями щодо окремих функцій.

Розробники материнських плат комп'ютерів повинні прийняти ряд рішень, наприклад про кількість роз'ємів для карт розширення PCIe, роз'ємів DIMM, портів USB і SATA, які повинні бути включені до складу конкретної моделі материнської плати. Ці рішення приймаються на основі цільової демографічної групи споживачів, будь то геймери, бізнес-користувачі або економні домашні користувачі.

Як приклад материнської плати візьмемо ASUS Prime X570-Pro (табл. 4.4). Це високопродуктивна плата, призначена для ігрових додатків і підтримує ігрові функції, такі як розгін процесора. Розгін – це збільшення тактової частоти процесора та інших компонентів системи з метою підвищення продуктивності.

Розгін також збільшує тепловиділення і може призвести до нестабільної роботи пристрою, якщо він працює на надмірній частоті.

Таблиця 4.4. *Приклад технічних характеристик материнської плати*

Компонент	Характеристика	Примітки
Процесор	Гніздо AMD AM4, сумісне з процесорами AMD Ryzen™ третього покоління	Гніздо містить 1331 контакт. Процесор взаємодіє безпосередньо з системною пам'яттю DDR4. Є інтерфейс PCI 4.0 x16, що безпосередньо з'єднує процесор з графічним процесором
Чіпсет	AMD X570. 16 ліній PCIe 4.0. 6 портів SATA, 6 Гбіт/с	Інтерфейс між процесором і чіпсетом: PCIe 4.0 x4

Таблиця 4.4. *(продовження)*

Компонент	Характеристика	Примітки
Графічна карта	До 3 роз'ємів PCIe 4.0 x16	Можлива паралельна робота декількох графічних процесорів через масштабований інтерфейс обміну даними Nvidia **Scalable Link Interface (SLI)** або за допомогою технології AMD Crossfire
Роз'єми розширення	3 роз'єми PCIe 4.0 x16. 3 роз'єми PCIe 4.0 x1	Роз'єми x16 можна налаштувати як один роз'єм x16, два роз'єми x8 або два роз'єми x8 плюс один роз'єм x4.
Системна пам'ять	4 двоканальних роз'єми для модулів DIMM DDR4 3200 МГц із загальною ємністю до 128 Гбайт	Можна встановити до чотирьох модулів DDR4 об'ємом до 32 Гбайт кожен. Можливий розгін до 5100 МГц.
Інтерфейси накопичувачів	2 роз'єми M.2. 6 роз'ємів SATA, 6 Гбіт/с	Роз'єми M.2 використовують інтерфейс PCI 4.0 для підключення швидкодіючих твердотільних накопичувачів. Роз'єми SATA призначені для підтримки традиційних дискових накопичувачів.
Ethernet	1 порт Gigabit Ethernet	Високошвидкісне підключення до мережі Ethernet
USB	2 роз'єми USB 2.0. 1 роз'єм USB 3.2, покоління 1 (5 Гбіт/с). 1 роз'єм USB 3.2, покоління 2 (10 Гбіт/с)	Кожен роз'єм підтримує кілька портів USB. Порти USB 2.0 призначені для клавіатури та миші. Порти USB 3.2 призначені для швидких периферійних пристроїв, таких як зовнішні накопичувачі.

Цей приклад призначений для того, щоб дати певне уявлення про характеристики комп'ютерів споживчого класу у верхньому ціновому сегменті станом на 2021 р.



Якщо ви збираєтеся придбати комп'ютер, можете використовувати інформацію з цього розділу, щоб стати більш поінформованим споживачем.

Резюме

Цей розділ розпочався з введення в пристрій елементарної комірки пам'яті комп'ютера, що складається з польового МОН-транзистора і конденсатора. Ми вивчили схему, що реалізує бітову комірку динамічної пам'яті DRAM. Ми розглянули архітектуру модулів пам'яті DDR5 і роботу багатоканальних контролерів пам'яті. Були представлені інші типи пристроїв вводу-виводу і окремо розглянуті високошвидкісні диференціальні послідовні з'єднання, широко використовувані в таких технологіях, як PCIe, SATA, USB, і відеоінтерфейси.

Були описані популярні відеостандарти, включаючи VGA, DVI, HDMI і DisplayPort. Ми також розглянули мережеві технології Ethernet і Wi-Fi. Потім обговорили стандартні інтерфейси комп'ютерної периферії, включаючи клавіатуру і мишу. Глава завершилася описом приблизних характеристик сучасної материнської плати з виділенням деяких цікавих особливостей.

Завдяки інформації, представленій в цьому розділі, у вас має скластися чітке уявлення про компоненти сучасних комп'ютерів, починаючи з рівня технічних характеристик і закінчуючи технологіями, що використовуються при реалізації схем.

У наступному розділі ми розглянемо функції більш високого рівня, які повинні бути реалізовані в комп'ютерних системах, такі як введення-виведення для накопичувачів, обмін даними по мережі і взаємодія з користувачами. Ми вивчимо рівні програмного забезпечення, що реалізують ці функції, починаючи з рівня набору інструкцій і регістрів процесора. Буде розглянуто ряд ключових аспектів операційних систем, в тому числі початкове завантаження, багатопотоковість і багатопроцесорність.

Вправи

1. Складіть схему реалізації логічного елемента І-НЕ, використовуючи дві пари МОН-транзисторів, об'єднаних в КМОН-структури. На відміну від схем вентилів на основі і-р-і-транзисторів, для цієї схеми не потрібні резистори.
2. 16-гігабітна інтегральна схема пам'яті DRAM має два входи вибору групи банків, два входи вибору банку і 17 входів вибору адрес рядків. Скільки бітів у кожному рядку банку в цьому пристрої?

5

Апаратно-програмний інтерфейс

Більша частина програмного забезпечення комп'ютерів написана не на рівні інструкцій процесора, тобто не мовою асемблера. Майже всі додатки, з якими ми щодня працюємо, написані тією чи іншою мовою програмування високого рівня на основі вже існуючих бібліотек можливостей, які розробники додатків використовували в процесі розробки програмного забезпечення. Застосовувані на практиці середовища програмування, що включають у себе мови високого рівня та пов'язані з ними бібліотеки, пропонують безліч сервісів, зокрема **функції вводу-виводу** дискових накопичувачів, обмін даними мережею та взаємодію з користувачами, які легко доступні з програмного коду.

У цьому розділі описуються рівні програмного забезпечення, що реалізують ці функції, починаючи з інструкцій процесора в драйверах пристроїв. Тут буде розглянуто низку ключових аспектів операційних систем, включно з початковим навантаженням, плануванням процесів, багатопотоковістю та багатопроцесорністю.

Прочитавши цей розділ, ви отримаєте уявлення про сервіси, що надаються операційними системами і функціями **базової системи введення виведення (BIOS)** і **єдиного розширюваного інтерфейсу вбудованого ПЗ** (Unified Extensible Firmware Interface, UEFI). Ви дізнаєтеся, як потоки обчислень функціонують на рівні процесора і як кілька процесорних ядер координують свої дії в комп'ютерній системі. Ви також отримаєте загальне уявлення про процес безпечного завантаження операційної системи, починаючи з виконання першої інструкції.

Ми розглянемо такі теми:

- драйвери пристроїв;
- базова система введення-виведення;
- процес завантаження операційної системи;
- операційні системи;
- процеси та потоки;
- багатопроцесорність.

Технічні вимоги

Файли для цієї глави, включно з відповідями до вправ, доступні за адресою https://github.com/PacktPublishing/Modern-Computer-Architecture-and-OrgяnizationSecond-Edition.

Драйвери пристроїв

Драйвер пристрою надає програмним додаткам стандартизований інтерфейс для взаємодії з периферійними пристроями, такими як дискові накопичувачі. Це позбавляє розробника додатка від необхідності розуміти і реалізовувати всі технічні деталі, необхідні для належної роботи пристроїв кожного типу. Драйвери пристроїв також забезпечують необхідну координацію, коли додатки, написані різними розробниками, намагаються одночасно отримати доступ до одного й того самого пристрою. Більшість драйверів пристроїв дають змогу кільком додаткам безпечно й ефективно взаємодіяти з кількома екземплярами відповідних периферійних пристроїв.

На найнижчому рівні код драйвера пристрою надає програмні інструкції, які керують обміном даними з периферійним пристроєм, включно з обробленням переривань, ініційованих запитами на обслуговування, що надсилаються в пристрій. Драйвер пристрою керує роботою апаратних ресурсів процесора, периферійного пристрою та інших компонентів системи, таких як чипсет процесора.

У комп'ютерних системах, що підтримують привілейовані режими виконання, драйвери пристроїв зазвичай працюють із підвищеним рівнем привілеїв. Код, що працює з підвищеним рівнем привілеїв, які зазвичай обмежені операційною системою та драйверами пристроїв, має повний доступ до можливостей процесора та периферійних пристроїв. Цей рівень гарантує надання доступу до ресурсів введення-виведення, недоступних для коду з меншими привілеями. Тільки довірений код може безпосередньо взаємодіяти з цими інтерфейсами. Якби код непривілейованого додатка міг одержати доступ до апаратного інтерфейсу периферійного пристрою, то програмна помилка, що спричинила неправильну поведінку пристрою, могла б безпосередньо вплинути на всі додатки, які намагаються використовувати цей пристрій. Дії, пов'язані з перемиканням потоку виконання інструкцій і даних між непривілейованим кодом користувача і привілейованим кодом драйвера, обговорюватимуться в *розділі 9*.

Драйвери, які виконуються з підвищеними привілеями, називаються **драйверами режиму ядра. Ядро** – це центральний елемент операційної системи, що слугує інтерфейсом між апаратними засобами комп'ютера і функціями операційної системи вищого рівня, такими як планувальник. Як ми дізналися з *розділу 3*, для доступу до пристроїв введення-виведення використовуються два основні механізми: з розподілом пам'яті та з розподілом за портами. У сучасних комп'ютерах переважає введення-

виведення з розподілом пам’яті, проте в деяких архітектурах, як-от x86, як і раніше, підтримують і використовують введення-виведення з розподілом за портами. В архітектурі x86 багато сучасних периферійних пристроїв надають системний інтерфейс, що поєднує обидва механізми введення-виведення.

Інструменти програмування для сучасних операційних систем, таких як Linux і Windows, надають ресурси для розроблення драйверів пристроїв, здатних взаємодіяти з периферійними пристроями, використовуючи методи введення-виведення як із розподілом за портами, так і з розподілом пам’яті. Встановлення драйвера пристрою в цих операційних системах вимагає підвищених привілеїв, але користувачам драйвера такі привілеї не потрібні.

Драйвери сучасних периферійних пристроїв можуть бути доволі складними і важкими для розуміння тими, хто не надто добре обізнаний з апаратною частиною і вбудованим ПЗ таких пристроїв, однак деякі пристрої колишніх поколінь доволі прості. Одним із прикладів таких пристроїв є паралельний порт принтера, що був запроваджений у ранніх моделях персональних комп’ютерів і протягом багатьох років залишався стандартним компонентом ПК. Сучасні комп’ютери зрідка оснащуються цими інтерфейсами, проте недорогі карти розширення з паралельними портами залишаються легкодоступними, а сучасні операційні системи забезпечують підтримку драйверів для цих інтерфейсів. Любителі електроніки часто використовують паралельний порт як простий інтерфейс для під’єднання до НК під керуванням Linux або Windows зовнішніх схем, які використовують 5-вольтові цифрові сигнали **транзисторно-транзисторної логіки** (ТТЛ).

У наступному розділі буде розглянуто деталі інтерфейсу паралельного порту на рівні драйвера пристрою.

Паралельний порт

Програмний інтерфейс для паралельного порту принтера на ПК складається з трьох 8-розрядних регістрів, від самого початку прив'язаних до послідовних номерів портів введення-виведення, починаючи з шістнадцяткового значення 378. Цей набір портів забезпечує роботу інтерфейсу для системного принтера № 1, що позначається як LPT1 на IBM PC-сумісних комп'ютерах під керуванням MS-DOS і Windows. Сучасні ПК можуть зіставляти паралельний порт з іншим діапазоном портів введення-виведення під час ініціалізації шини **Peripheral Component Interconnect (PCI)**, але в іншому, порівняно з ранніми ПК, робота цього інтерфейсу не змінилася.

Драйвери пристроїв для паралельного порту в сучасних комп'ютерах виконують ті ж функції, використовуючи ті ж інструкції, що і в ранніх ПК. У цьому випадку ми припустимо, що порт принтера зіставлено з традиційним діапазоном портів введення-виведення в 64-розрядній версії Linux.

Для взаємодії з апаратною частиною паралельного порту процесор x86 виконує інструкції in і out, що забезпечують читання з портів введення-виведення і запис у них.

Якщо припустити, що драйвер паралельного порту встановлений та ініціалізований, то користувацький застосунок може викликати функцію драйвера для читання стану ліній передавання даних паралельного порту та запису в них.

Наступні інструкції драйвера зчитують дискретні рівні напруги, які присутні на восьми лініях передавання даних паралельного порту, і зберігають отримане 8-бітове значення в регістрі al процесора:

```
mov edx,0x378
in al,dx
```

Мовою асемблера x86 інструкції з двома операндами записуються у вигляді оп-код адресат, джерело. У цьому прикладі використано регістри al, edx і dx процесора. Регістр al містить молодші 8 біт 32-розрядного регістра eax, а dx – молодші 16 біт 32-розрядного регістра edx. Ця послідовність інструкцій завантажує безпосереднє значення 8x378 у регістр edx, після чого зчитує в регістр al 8-бітове значення даних порту, номер якого міститься в dx.

Вихідний код мовою C, що дає змогу згенерувати зазначені вище інструкції на асемблері:

```
char input_byte;
input_byte = inb(0x378);
```

Функцію inb, призначену для виконання введення 8-бітного значення з порту введеннявиведення, надає операційна система Linux. Цей код буде функціонувати належним чином тільки в тому випадку, якщо він виконується в цій операційній системі з підвищеними привілеями. Додаток, запущений на рівні привілеїв користувача, під час спроби виконати ці інструкції зазнає невдачі, оскільки він не має повноважень на виконання операцій введення-виведення для порту безпосередньо.

Інструкції для запису байта в регістр даних паралельного порту і тим самим встановлення стану восьми дискретних вихідних сигналів показано в такому фрагменті коду:

```
mov edx,0x378
movzx eax,BYTE PTR [rsp+0x7]
out dx,al
```

Цей код переносить у регістр edx номер порту, потім завантажує в регістр eax змінну зі стека. rsp – це 64-бітний покажчик стека. Значення rsp є 64-бітовим, оскільки

драйвер працює в 64-розрядній версії Linux. Опкод mvzx означає "переміщення з доповненням нулями", тобто ця інструкція переміщає 8-бітове значення даних (з позначенням BYTE PTR), що зберігається за адресою rsp+8x7, у молодші 8 біт 32-бітного регістра eax, заповнюючи 24 біти, що залишаються, у регістрі eax нулями. Остання інструкція записує байт із регістра al за номером порту, що зберігається в регістрі dx.

Вихідний код мовою C, що генерує ці інструкції:

```
char output_byte = 0xA5;
outb(output_byte,0x378);
```

Аналогічно inb, функція outb, що надається ОС Linux, дає змогу драйверам пристроїв записувати 8-бітові значення в заданий порт введення-виведення.

Цей приклад демонструє механізм взаємодії між програмою, що виконується процесором, і апаратними регістрами периферійних пристроїв на найнижчому рівні роботи драйвера пристрою. Драйвери для складніших пристроїв у системах x86 зазвичай поєднують метод введення-виведення з розподілом по портах, показаний у попередніх прикладах, із методом введення-виведення з розподілом пам'яті, що звертається до інтерфейсу пристрою за допомогою операцій читання та запису, з погляду інструкцій процесора ідентичних до звернень до пам'яті.

У цих прикладах представлено методи доступу до апаратних засобів, що використовуються драйверами в оригінальній архітектурі паралельної шини PCI. У наступному розділі розглядаються функції, завдяки яким традиційні драйвери PCI можуть коректно працювати на сучасних комп'ютерах з PCIe без змін, повноцінною мірою використовуючи переваги технології високошвидкісної послідовної передачі даних PCIe.

Драйвери пристроїв PCIe

Як було показано в попередньому розділі, PCIe використовує високошвидкісні послідовні з'єднання як каналів зв'язку між процесором і периферійними пристроями з підтримкою PCIe. Можливо, вам цікаво, які дії повинен виконувати драйвер пристрою для взаємодії з такими потужними апаратними засобами. Проста відповідь полягає в тому, що драйверам не потрібно робити нічого особливого, щоб повною мірою скористатися широкими можливостями шини PCIe. Технологію PCIe розробили з урахуванням вимог програмної сумісності з паралельною шиною PCI, яку використовували в ПК 1990-х років. Драйвери пристроїв, написані для PCI, як і раніше коректно працюють на комп'ютерах, що використовують послідовну шину PCIe. Завдання переведення інструкцій введення-виведення процесора, таких як in і out, в операції послідовного передавання даних, яке необхідно вирішити для обміну

даними з пристроями PCIe, прозоро виконують підсистеми PCIe у процесорі, чипсеті та пристроях із підтримкою PCIe.

Пристрої PCI і PCIe автоматично виконують операцію налаштування під час запуску системи і під час "гарячого" під'єднання пристрою до системи, що працює. **"Гаряче" під'єднання** – це встановлення апаратного пристрою в систему, яка ввімкнена.

Після завершення налаштування інтерфейс пристрою стає доступний операційній системі. Інтерфейс між периферійним пристроєм PCI або PCIe і процесором може містити будь-яку комбінацію таких механізмів обміну даними:

- один або кілька діапазонів портів введення-виведення;
- одна або кілька областей адрес пам'яті, що підтримують введення-виведення з розподілом пам'яті;
- підключення до обробника переривань процесора.

Процедура налаштування інтерфейсу застосовується як до драйверів PCI, так і до PCIe, даючи змогу застарілим драйверам PCI коректно працювати в системах із PCIe. Звісно ж, інтерфейси фізичних карт пристрою з паралельною шиною PCI і пристрої з послідовною шиною PCIe сильно різняться, тому самі карти не є взаємозамінними, оскільки використовують шини на основі різних технологій. Роз'єми шин PCIe навмисно виконані такими, що відрізняються від роз'ємів PCI, щоб запобігти випадковому встановленню пристроїв PCI в роз'єми PCIe, і навпаки.

Масова передача даних у периферійні пристрої та з них зазвичай заснована на технології **прямого доступу до пам'яті** (direct memory access, DMA) – як для PCI, так і для PCIe. У системах із підтримкою PCIe операції DMA повною мірою використовують високі швидкості передавання даних, досяжні за допомогою багатолінійних послідовних з'єднань, передаючи дані через інтерфейс зі швидкістю, близькою до теоретичної максимальної швидкості, яку може підтримати кожен одно- або багатолінійний канал послідовного зв'язку. Технологічний розвиток, що викликав витіснення застарілої технології паралельної шини PCI значно більш швидкодіючою технологією багатолінійного послідовного передавання даних PCIe за умови збереження повної сумісності з драйверами пристроїв, був вельми примітним явищем.

Структура драйверів пристроїв

Драйвер пристрою – це програмний модуль, який реалізує набір заздалегідь визначених функцій, що дають змогу операційній системі пов'язувати драйвер із сумісними периферійними пристроями та здійснювати контрольований доступ до цих пристроїв. Це дає можливість системним процесам і користувацьким додаткам виконувати операції введення-виведення на пристроях, які використовуються спільно. У цьому розділі подано короткий огляд функцій, які мають бути реалізовані в драйвері

пристрою Linux для використання розробниками додатків. У цьому прикладі до імен функцій як префікс додається фіктивне ім'я пристрою mydevice. Приклад написаний мовою програмування С.

Наступні функції виконують ініціалізацію та завершення роботи драйвера:

```
int mydevice_init(void);
void mydevice_exit(void);
```

Операційна система викликає mydevice_init для налаштування драйвера пристрою під час запуску системи або пізніше, якщо пристрій додається до системи за допомогою «гарячого» підключення. Функція mydevice_init повертає цілочисельний код, що вказує, чи була ініціалізація успішною, або, у разі невдачі, код помилки, що виникла. На успішну ініціалізацію драйвера вказує повернення нуля.

Коли драйвер більше не потрібен, наприклад при завершенні роботи системи, або коли пристрій від'єднують під час роботи системи, система викликає функцію mydevice_exit, щоб повністю припинити доступ до пристрою і звільнити всі системні ресурси, виділені для драйвера.

Наступні дві функції дозволяють системним процесам і користувацьким додаткам ініціювати і завершувати сеанси зв'язку з пристроєм:

```
int mydevice_open(struct inode *inode, struct file *filp);
int mydevice_release(struct inode *inode, struct file *filp);
```

Функція mydevice_open намагається ініціювати доступ до пристрою і повідомляє про будь-які помилки, які можуть виникнути при цьому. Параметр inode – це покажчик на структуру даних, що містить інформацію, необхідну для доступу до певного файлу або іншого пристрою. Параметр fiip – це покажчик на структуру даних, що містить інформацію про відкритий файл. У підсистемі введення-виведення Linux всі типи пристроїв послідовно представлені у вигляді файлів, навіть якщо сам пристрій за своєю суттю не базується на файлах. Ім'я параметра filp – це скорочення від file pointer (покажчик файлу). Всі функції, що працюють з даним файлом, отримують покажчик на цю структуру в якості вхідних даних. Серед інших деталей структура filp вказує, чи відкритий файл для читання, запису або для того й іншого.

Функція mydevice_release закриває пристрій або файл і звільняє всі ресурси, виділені при виклику функції mydevice_open.

Після успішного звернення до функції mydevice_open код програми може почати зчитування даних з пристрою або запис на нього. Функції, що виконують ці операції:

```
ssize_t mydevice_read(struct file *filp, char *buf,
    size_t count, loff_t *f_pos);
```

```
ssize_t mydevice_write(struct file *filp, const char *buf,
    size_t count, loff_t *f_pos);
```

Функція mydevice_read зчитує дані з пристрою або файлу і передає їх у буфер у просторі пам'яті програми. Параметр count вказує запитуваний обсяг даних, а параметр f_pos задає зміщення від початку файлу, з якого слід почати читання даних. Параметр buf – це адреса місця призначення даних. Кількість фактично зчитаних байтів (яка може бути меншою за запитане число) надається функцією у вигляді значення, що повертається, з типом даних ssize_t.

Функція mydevice_write використовує більшість тих самих параметрів, що й mydevice_read, за винятком того, що параметр buf оголошується константою (const), оскільки функція mydevice_write зчитує дані, починаючи з адреси пам'яті, зазначеної параметром buf, і записує дані у файл або пристрій.

Один із цікавих моментів у реалізації цих функцій полягає в тому, що привілейований код драйвера не може (або, принаймні, не повинен, якщо операційна система дозволяє це) безпосередньо звертатися до користувацької пам'яті. Це зроблено для запобігання випадковому або навмисному виконанню кодом драйвера зчитування або запису в неприпустиму область пам'яті, таку як простір ядра.

Для того щоб уникнути цієї потенційної проблеми, операційна система надає спеціальні функції – copy_to_user і copy_from_user, які слід використовувати в драйверах для доступу до користувацької пам'яті. Перед копіюванням даних ці функції виконують необхідні дії для перевірки адрес користувацького простору, зазначених у виклику функції.

У цьому розділі дано короткий вступ в операції апаратного інтерфейсу, що виконуються драйверами пристроїв, і представлена загальна структура драйвера пристрою.

Під час увімкнення комп'ютера перед завантаженням операційної системи та ініціалізацією її драйверів для проведення низькорівневого самотестування та налаштування системи необхідно запустити вбудоване ПЗ. У наступному розділі представлено вступ до коду, який виконується під час увімкнення комп'ютера перед іншими програмами: базова система вводу-виводу.

Базова система введення-виведення (BIOS)

Базова система введення-виведення комп'ютера (Basic Input/Output System, BIOS) містить код, який виконується під час запуску системи. На зорі появи персональних комп'ютерів BIOS надавала собою набір програмних інтерфейсів, які допомагали абстрагуватися від деталей інтерфейсів периферійних пристроїв, таких як клавіатури та дисплеї.

У сучасних ПК під час запуску BIOS виконує тестування системи та налаштування периферійних пристроїв. Після завершення цих дій процесор (під керуванням програм) взаємодіє з периферійними пристроями безпосередньо, без подальшого втручання BIOS.

У ранніх ПК код BIOS зберігався в чипі **постійної пам'яті** (постійному запам'ятовуючому пристрої, ПЗУ – read-only memory, ROM) на материнській платі. Цей код був жорстко запрограмований і не міг бути змінений. Сучасні материнські плати зазвичай зберігають BIOS материнської плати в перепрограмованому пристрої флеш пам’яті. Це дає змогу встановлювати оновлення BIOS для додавання нових функцій або усунення проблем, виявлених у більш ранніх версіях вбудованого ПЗ. Процес оновлення вмісту BIOS зазвичай називають *перезаписом BIOS*.

Один із недоліків перепрограмування BIOS полягає в тому, що ця можливість дозволяє впроваджувати в систему шкідливий код шляхом запису його у флеш-пам'ять BIOS. У разі успішної реалізації атаки цього типу шкідливий код виконується під час кожного запуску комп'ютера. На щастя, успішні атаки на вбудоване ПЗ BIOS виявилися досить рідкісними.

Оскільки BIOS перебирає на себе керування під час запуску системи, однією з перших її дій є запуск **самотестування під час увімкнення живлення** (power-on self-test, POST) для ключових компонентів системи. Під час виконання POST-процедури BIOS намагається взаємодіяти із системними компонентами, включно з клавіатурою, дисплеєм і завантажувальним пристроєм, у ролі якого зазвичай виступає дисковий накопичувач. Комп'ютер може бути оснащений високопродуктивним графічним процесором, однак відеоінтерфейс, який використовується BIOS під час запуску, може виявитися досить примітивним і підтримувати лише виведення тексту.

BIOS використовує відеоінтерфейс та інтерфейс клавіатури для відображення будь-яких помилок, виявлених під час тестування системи, і дає змогу користувачеві змінювати збережені налаштування. Клавіатура та відеоінтерфейси, що надаються BIOS, дають змогу виконати початкове налаштування комп'ютера, у якому ще немає завантажувального пристрою.

Якщо відеоінтерфейс працює неправильно, BIOS не зможе відобразити інформацію, пов'язану з помилкою. У цій ситуації BIOS намагається використовувати динамік ПК, за його наявності, щоб указати на помилку, видаючи послідовність звукових сигналів. документація по материнській платі містить інформацію про типи помилок, на які вказує кожна така послідовність. Деякі материнські плати мають цифровий дисплей для виведення відомостей про помилки під час тестів POST та іншої інформації про стан.

Залежно від конфігурації системи ініціалізацією пристроїв PCIe під час запуску системи керує або BIOS, або операційна система. Після успішного завершення процесу налаштування всім пристроям PCIe призначено сумісні діапазони портів

введення-виведення, діапазони введення-виведення для режиму з розподілом пам'яті та номери переривань.

У міру виконання процедури запуску операційна система визначає відповідний драйвер для зв'язку з кожним периферійним пристроєм на основі інформації про виробника і пристрою, що надається периферійним пристроєм через інтерфейс PCIe. Після успішної ініціалізації драйвер напряму взаємодіє з кожним периферійним пристроєм для виконання операцій введення-виведення за запитом. Системні процеси та користувацькі додатки викликають набір стандартизованих функцій драйверів, представлених у попередньому розділі, для отримання доступу до пристрою, виконання операцій читання і запису та закриття пристрою.

Однією з поширених процедур, пов'язаних з BIOS, які виконують користувачі, є вибір порядку завантаження серед доступних пристроїв зберігання даних. Наприклад, ця функція дає змогу налаштувати систему на першу спробу завантаження з оптичного диска, що містить образ операційної системи, якщо такий диск є в дисководі. Якщо завантажувальний оптичний диск не знайдено, система може спробувати завантажитися з основного дискового накопичувача. Для завантаження операційної системи можна налаштувати й розташувати в пріоритетному порядку кілька пристроїв зберігання даних.

Для виклику режиму налаштування BIOS іноді призначається певна клавіша, наприклад <Esc> або функціональна клавіша <F2>, яку слід натиснути на ранній стадії процесу завантаження. Назва відповідної клавіші зазвичай відображається на екрані незабаром після ввімкнення живлення. Під час входу в режим налаштування BIOS параметри відображаються у форматі меню. Ви можете вибрати один із доступних екранів, щоб змінити параметри, пов'язані з такими функціями, як порядок пріоритетів завантаження. Після внесення змін до параметрів користувачеві надається можливість зберегти ці зміни в **енергонезалежній пам'яті** (nonvolatile memory, NVM) і відновити процес завантаження. При цьому слід дотримуватися обережності, оскільки внесення неналежних змін у налаштування BIOS може призвести до того, що комп'ютер не завантажиться.

За час, що минув з моменту появи IBM PC, можливості реалізації BIOS значно розширилися. У міру розвитку архітектури ПК і додавання підтримки 32-розрядних, а потім і 64-розрядних операційних систем застаріла архітектура BIOS не встигала за потребами нових, більш зручних систем. Великі учасники галузі виступили з ініціативою визначити архітектуру вбудованого програмного забезпечення системи, яка допомогла б позбутися обмежень BIOS. Результатом цих зусиль став стандарт UEFI, який замінив традиційні можливості BIOS у сучасних материнських платах.

Єдиний розширюваний інтерфейс вбудованого ПЗ (UEFI)

Єдиний розширюваний інтерфейс вбудованого ПЗ (Unified Extensible Firmware Interface, UEFI) – це випущений у 2007 р. стандарт, який визначає архітектуру вбудованого ПЗ, що реалізує функції, які надають застарілі BIOS, і додає низку істотних поліпшень. Як і у випадку з BIOS, UEFI містить код, що виконується відразу після запуску системи.

Введення UEFI має на меті кілька цілей, включно з додаванням підтримки завантажувальних дискових накопичувачів об'ємом понад 2 терабайти (Тбайт), скороченням часу запуску і підвищенням безпеки процесу завантаження. UEFI надає кілька функцій, які під час увімкнення та правильного використання істотно знімають імовірність випадкового або зловмисного пошкодження вбудованого ПЗ, що зберігається у флеш-пам'яті UEFI.

На додаток до можливостей застарілих реалізацій BIOS, описаних раніше, UEFI підтримує такі функції.

- **Додатки UEFI** – це модулі виконуваного коду, що зберігаються у флеш-пам'яті UEFI. Додатки UEFI розширюють спектр можливостей, доступних у перед завантажувальному середовищі материнської плати, і в деяких випадках, надають сервіси для використання операційними системами під час виконання. Одним із прикладів додатків UEFI є оболонка UEFI, яка являє собою інтерфейс командного рядка для взаємодії з процесором і периферійними пристроями. Оболонка UEFI підтримує запити даних від пристроїв і дає змогу змінювати параметри конфігурації, що зберігаються в енергонезалежній пам'яті.
- Іншим прикладом програми UEFI є GNU **GRand Unified Bootloader (GRUB)**. GRUB підтримує багато завантажувальні конфігурації, представляючи меню, з якого користувач може вибрати один із кількох доступних образів операційної системи для завантаження під час запуску системи.
- **Архітектурно-незалежні драйвери пристроїв** надають UEFI доступ до незалежних від процесора реалізацій драйверів пристроїв. Завдяки цьому можна використовувати єдину реалізацію вбудованого ПЗ UEFI на таких різнопланових архітектурах, як x86 і процесори **Advanced RISC Machine (ARM)**. Архітектурно незалежні драйвери UEFI зберігаються у формі байтового коду, який інтерпретується вбудованим ПЗ, що відповідає типу процесора. Під час завантаження ці драйвери забезпечують взаємодію UEFI з периферійними пристроями, такими як графічні карти і мережеві інтерфейси.
- **Безпечне завантаження** – використання криптографічних сертифікатів, які гарантують, що під час запуску системи виконуються тільки схвалені драйвери пристроїв і завантажувачі операційної системи. Ця функція перевіряє цифровий

підпис кожного компонента вбудованого ПЗ, перш ніж дозволити його виконання. Така перевірка захищає від різноманітних шкідливих атак, націлених на вбудоване ПЗ.

- **Прискорення завантаження** досягається за рахунок паралельного виконання операцій, які в BIOS виконувалися послідовно. На практиці завантаження відбувається настільки швидко, що багато реалізацій UEFI не пропонують користувачеві можливість натискати клавішу під час завантаження, оскільки очікування відповіді веде до затримки запуску системи. Натомість операційні системи, такі як Windows, забезпечують доступ до налаштувань UEFI, пропонуючи користувачеві можливість запитувати цей доступ під час роботи операційної системи з подальшим перезавантаженням для відображення екрана налаштувань UEFI.

UEFI не просто замінює функції старого BIOS. Це мініатюрна операційна система, що підтримує розширені можливості, наприклад, дає змогу віддаленому фахівцеві використовувати мережеве під'єднання для усунення неполадок на комп'ютері, який не завантажується.

Після тестів POST, низькорівневого налаштування системних пристроїв і визначення відповідного завантажувального пристрою на основі порядку виконання завантаження система починає процес завантаження операційної системи.

Процес завантаження операційної системи

Процедура завантаження образу системи залежить від структури (формату) розділів на пристрої зберігання даних, де записано цей образ, і функцій безпеки, що застосовуються під час завантаження. Мета процесу завантаження – запустити систему після ввімкнення живлення та ініціалізувати операційну систему, залишивши комп'ютер у відомому стані та готовим до виконання корисної роботи.

З початку 1980-х років стандартний формат розділів диска називався **основним завантажувальним записом** (master boot record, MBR). Розділ MBR має завантажувальний сектор, розташований на логічному початку відведеного для нього простору. Цей завантажувальний сектор містить інформацію, що описує логічні розділи пристрою. Кожен розділ містить файлову систему, організовану у вигляді деревоподібної структури каталогів і файлів усередині них.

Через фіксований формат структур даних MBR запам'ятовуючий пристрій MBR може містити максимум чотири логічні розділи та мати об'єм не більш ніж 2 Тбайт, тобто не більше 2^{32} 512-байтових секторів для розміщення даних. Ці обмеження стали вельми відчутними, коли обсяг доступних на ринку дискових накопичувачів перетнув межу в 2 Тбайт. Для вирішення цих проблем і паралельно з розробкою UEFI був розроблений новий формат розділів – **таблиця розділів GUID** (GUID Partition Table, GPT), де **GUID** означає **глобальний унікальний ідентифікатор** (Global Unique

IDentifier). Цей формат усунув обмеження щодо об'єму накопичувача і кількості розділів, надавши при цьому деякі додаткові удосконалення.

Накопичувач, розмічений за форматом GPT, має максимальний розмір 2^{64} 512-байтових секторів, що вміщають понад 8 млрд терабайт даних. За звичайного налаштування GPT підтримує до 128 розділів на накопичувач. Тип кожного розділу вказується 128-бітовим ідентифікатором GUID, що дає змогу в майбутньому визначати фактично необмежену кількість нових типів розділів. Більшості користувачів не потрібна велика кількість розділів на одному накопичувачі, тому найбільш очевидною перевагою GPT для користувачів є підтримка накопичувачів більшого обсягу.

Процес завантаження для материнських плат з BIOS і з UEFI має деякі відмінності, описані в наступних розділах.

Завантаження при використанні BIOS

На материнській платі з BIOS після тестів POST і налаштування пристроїв PCIe BIOS ініціює процес завантаження. BIOS робить спробу виконати навантаження з першого пристрою в налаштованій послідовності пріоритетів. Якщо виявлено допустимий пристрій, вбудоване ПЗ зчитує із завантажувального сектора MBR невеликий фрагмент виконуваного коду, званий *завантажувачем*, і передає йому керування. На цьому вбудоване ПЗ BIOS завершує свою роботу і більше не використовується під час функціонування системи. Завантажувач ініціює процес завантаження і запуску операційної системи.

Якщо на материнській платі з BIOS використовується *диспетчер завантаження*, код завантажувального сектора MBR запускає цей диспетчер замість безпосереднього виконання завантаження операційної системи. Диспетчер завантаження (наприклад, GRUB) відображає список, з якого користувач вибирає необхідний образ операційної системи. Вбудоване ПЗ BIOS не має інформації про багато завантажувальні конфігурації, і процес вибору операційної системи в диспетчері завантаження відбувається без участі BIOS.

БАГАТОЗАВАНТАЖУВАЛЬНА КОНФІГУРАЦІЯ І ПРІОРИТЕТИ ЗАВАНТАЖЕННЯ



Багато завантажувальна конфігурація в диспетчері завантаження дає змогу користувачеві вибрати потрібну операційну систему з меню додаткових варіантів. Це меню відрізняється від пріоритетного списку варіантів завантаження, підтримуваного BIOS, який дозволяє самій BIOS вибирати перший доступний образ операційної системи.

Завантаження під час використання UEFI

На материнській платі з UEFI після завершення тестів POST і налаштування пристроїв (способом, дуже схожим на відповідні дії BIOS) UEFI ініціює процес завантаження. Під час процедури запуску може відображатися диспетчер завантаження. Диспетчер завантаження UEFI, що є частиною вбудованого ПЗ UEFI, відображає меню, з якого користувач може вибрати необхідний образ операційної системи.

Якщо користувач не вибирає операційну систему з меню диспетчера завантаження протягом декількох секунд (або якщо меню диспетчера завантаження не відображається), UEFI намагається виконати навантаження з першого пристрою в налаштованій послідовності пріоритетів.

Вбудоване ПЗ UEFI зчитує виконуваний код диспетчера завантаження (який існує окремо від диспетчера завантаження UEFI) і файли завантажувача з налаштованих розташувань на системному диску, потім виконує ці файли в процесі запуску.

На наступній копії екрана показано частину системних **даних конфігурації завантаження** (boot configuration data, BCD), що зберігаються в системі Windows 10. Для відображення цієї інформації на своєму комп'ютері вам слід ввести команду bcdedit, використовуючи інтерфейс командного рядка з правами адміністратора.

```
C:\>bcdedit
Windows Boot Manager
--------------------
identifier {bootmgr}
device partition=\Device\HarddiskVolume1 path \EFI\MICROSOFT\BOOT\BOOTMGFW.EFI
...
Windows Boot Loader
-------------------
identifier {current} device partition=C:
path \WINDOWS\system32\winload.efi
...
```

У цьому прикладі **диспетчер завантаження Windows** знаходиться у файлі UEFI\MICROSOFT\BOOT\BOOTMGFW.EFI. Цей файл зазвичай зберігається в прихованому розділі диска і не завжди доступний для відображення в списках каталогів.

Завантажувач Windows ідентифікується як \WINDOWS\system32\winload.efi і розташований у каталозі c: \Windows\System32\winload.efi.

Імовірність зловмисної модифікації вбудованого ПЗ UEFI або програмних компонентів, що зберігаються на диску, які виконуються під час завантаження

системи, вимагає додаткового захисту, що забезпечується процесом довіреного завантаження.

Довірене завантаження

Метою процесу довіреного завантаження є виконання всієї послідовності завантаження з гарантією, що все програмне забезпечення, яке виконується, належним чином авторизоване і не було змінено. Завантажувальне ПЗ перевіряє достовірність вбудованого ПЗ і файлів програм, обчислюючи криптографічну хеш-функцію послідовності байтів, що становлять кожен програмний компонент, і підтверджуючи правильність підсумкового значення хеш-функції за допомогою цифрового підпису.

Криптографічна хеш-функція генерує "відбиток пальця" для блоку даних довільної довжини. Вихідне значення, що отримується за допомогою даної криптографічної хеш-функції, завжди має одну й ту саму довжину. Наприклад, 256-бітний алгоритм безпечного хешування **Secure Hashing Algorithm (SHA-256)** завжди дає 256-бітний результат незалежно від довжини вхідних даних.

Криптографічна хеш-функція забезпечує безпеку завдяки тому, що будь-яка спроба змінити дані, що використовуються цією функцією в якості вхідних даних, призведе до того, що вихідні дані хеш-функції також зміняться. Визначити набір модифікацій блоку вхідних даних, за яких хеш-функція видаватиме той самий результат, що й під час розрахунку за вихідним блоком даних, фактично нереально. У теорії можна створити модифікований набір даних, для якого алгоритм SHA-256 видаватиме той самий результат, що й початковий блок, однак необхідні для цього обчислення на найшвидшому доступному на сьогоднішній день суперкомп’ютери зайняли б більше часу, ніж решта терміну існування Землі. Ми обговоримо хеш-функції більш детально в *розділі 15*.

Для того щоб створити цифровий підпис, видавець програмного забезпечення повинен обчислити значення хеш-функції для збереженого в пам'яті образу вбудованого ПЗ або програмного компонента. Наступний крок – зашифрувати хеш-значення за допомогою закритого ключа. Цей закритий ключ шифрування пов'язаний із загальнодоступним ключем розшифрування. Зашифроване хеш-значення формує цифровий підпис, який зберігається разом із файлом вбудованого ПЗ або програмного компонента.

Відкритий ключ стає доступним для користувачів вбудованого ПЗ або програмного компонента. Цей ключ використовується для розшифрування зашифрованого хешзначення, яке потім можна порівняти з хеш-значенням, обчисленим на основі послідовності байт даних файлу під час завантаження. Ці два хеш-значення мають точно збігатися, що підтвердить факт незмінності файлу з моменту його підписання цифровим підписом. Як і у випадку з криптографічним алгоритмом хешування,

фактично неможливо змінити цифровий підпис таким чином, щоб приховати зміни в захищеному файлі даних.

Відкритий ключ не потрібно зберігати в таємниці, однак користувачі цього ключа повинні переконатися, що сам ключ отримано з надійного джерела і не був замінений зловмисником. Для того щоб гарантувати, що в процесі завантаження використовуються тільки довірені відкриті ключі, ці ключі зберігаються в апаратному сховищі, недоступному для неавторизованих користувачів і програмного забезпечення. Стандартний механізм для цього в НК називається **довіреним платформним модулем** (trusted platform module, TPM). По суті, TPM – це мікроконтролер, розроблений спеціально для захисту криптографічних ключів.

На нових комп'ютерах ключі TPM налаштовуються виробником системи для забезпечення безпечного завантаження системи з моменту її первісного ввімкнення.

На відміну від персональних комп'ютерів, більшість мобільних пристроїв використовують набагато простіший процес завантаження, який не задіює BIOS або UEFI. У наступному розділі розглядається процес завантаження мобільних пристроїв.

Мобільні пристрої

Більшість мобільних пристроїв, таких як смартфони, зазвичай не мають окремого вбудованого завантажувального ПЗ, подібного до BIOS або UEFI у ПК. Як ми бачили на прикладі 6502, під час подачі живлення такі пристрої виконують апаратне скидання процесора і починають виконання коду за вказаною адресою. Весь код у цих пристроях зазвичай міститься в енергонезалежній області пам'яті, такій як флеш-пам'ять.

Під час запуску мобільні пристрої виконують послідовність, аналогічну процесу завантаження ПК. Перевірка правильності роботи та ініціалізація периферійних пристроїв проводяться перед їх першим використанням. Завантажувачу на таких пристроях може знадобитися вибрати один з декількох розділів пам'яті, щоб визначити відповідний образ системи. Як і у випадку з UEFI, мобільні пристрої часто використовують функції безпеки під час процесу завантаження з метою переконатися, що завантажувач і образ операційної системи є справжніми, перш ніж дозволити продовження процесу завантаження.

Як на ПК, так і в мобільних пристроях запуск завантажувача є першим кроком процедури запуску операційної системи. Далі ми розглянемо запуск операційної системи.

Операційні системи

Операційна система – це багаторівневий пакет програмного забезпечення, що забезпечує створення середовища, у якому додатки виконують корисні функції, як-от

опрацювання текстів, виконання телефонних викликів або керування роботою автомобільного двигуна. Додатки, що працюють під керуванням операційної системи, виконують алгоритми, реалізовані у вигляді послідовностей інструкцій процесора, і в міру необхідності обмінюються даними з периферійними пристроями за допомогою операцій введення-виведення для виконання своїх завдань.

Операційна система надає стандартизовані програмні інтерфейси, які розробники додатків використовують для доступу до системних ресурсів, таких як потоки виконання процесора, файли на диску, введення з клавіатури або інших периферійних пристроїв і виведення на такі пристрої, як екран комп'ютера або прилади на приладовій панелі автомобіля.

Операційні системи можна розділити на дві широкі категорії – ті, що працюють у реальному часі, і ті, що працюють не в реальному часі:

- **Операційна система реального часу** (real-time operating system, RTOS) надає функції, що гарантують, що реакція на кодові дії відбудеться протягом певного часу.

  Процесори, що розв'язують такі завдання, як керування роботою автомобільного двигуна або кухонного приладу, зазвичай працюють під керуванням операційної системи реального часу. Це слугує гарантією того, що електричні та механічні компоненти, якими вони керують, отримують реакцію на будь-які зміни вхідних впливів протягом строго обмеженого часу.

- **Операційні системи, що працюють не в реальному часі**, не гарантують отримання відповіді на вхідні впливи в задані терміни. Замість цього такі системи намагаються виконати опрацювання якнайшвидше, навіть якщо іноді для завершення роботи потрібно багато часу.

ОПЕРАЦІЙНІ СИСТЕМИ РЕАЛЬНОГО ЧАСУ ПОРІВНЯНО З ЗВИЧАЙНИМИ ОПЕРАЦІЙНИМИ СИСТЕМАМИ



Операційні системи реального часу (ОСРЧ) не обов'язково є більш швидкими, ніж звичайні операційні системи, що працюють не в реальному часі. Звичайна операційна система може бути в середньому швидшою порівняно з ОСРЧ, але водночас ця система може іноді перевищувати тимчасові обмеження, встановлені для додатків, що виконуються в ОСРЧ. Мета ОСРЧ – ніколи не перевищувати граничний час відгуку.

Здебільшого операційні системи загального призначення, такі як Linux і Windows, не є операційними системами реального часу. Вони намагаються виконати призначені завдання, на кшталт завантаження файлу в текстовий редактор або обчислення

електронної таблиці, якомога швидше, хоча час виконання операції може сильно змінюватися залежно від інших завдань, які може виконувати система.

Деякі аспекти операційних систем загального призначення, зокрема виведення аудіо та відеосигналів, висувають особливі вимоги до роботи в реальному часі. Ми всі в той чи інший момент спостерігали збої у відтворенні відео, за якого зображення на екрані переривається і підвисає. Такий ефект є результатом невиконання вимог дисплея до пропускної здатності в реальному часі. Стільникові телефони висувають аналогічні вимоги до підтримки двостороннього обміну аудіоданими в реальному часі під час телефонних викликів.

Як на стандартних ПК, так і на мобільних пристроях, запуск операційної системи (звичайної або тієї, що працює в реальному часі), як правило, виконується з використанням аналогічної послідовності кроків. Завантажувач або завантажує ядро операційної системи в пам'ять, або просто переходить за адресою у віртуальній пам'яті, щоб почати виконання коду операційної системи. Після запуску ядро операційної системи виконує такі дії (хоча й не обов'язково в зазначеному порядку).

- Налаштування конфігурації процесора та інших системних пристроїв. Сюди належить налаштування будь-яких необхідних внутрішніх регістрів процесора і будь-яких пов'язаних із ними пристроїв керування введенням-виведенням, таких як чипсет.
- У системах, що використовують віртуальну пам'ять зі сторінковою організацією (яку буде представлено в *розділі 7*), ядро налаштовує модуль керування пам'яттю.
- Запуск системних процесів базового рівня, включаючи **планувальник** і **процес простою**. Планувальник керує послідовністю виконання потоків процесів. Процес простою містить код, який виконується, коли немає інших потоків, готових до запуску планувальником.
- Складання переліку драйверів пристроїв та їх прив'язка до кожного периферійного пристрою в системі. Для кожного драйвера виконується код ініціалізації, як обговорювалося раніше в цьому розділі.
- Налаштування та ввімкнення переривань. Після увімкнення переривання система починає виконувати операції введення-виведення з периферійними пристроями.
- Запуск системних сервісів. Ці процеси підтримують дії, не пов'язані з операційною системою (наприклад, роботу з мережами), а також постійно діючі встановлені функції (наприклад, веб-сервер).
- Для персональних комп'ютерів запускається процес користувацького інтерфейсу, який відображає екран входу в систему. Цей екран дає користувачеві

можливість ініціювати інтерактивний сеанс роботи з комп'ютером. На мобільних пристроях запускається додаток реального часу. Базова послідовність операцій для простого вбудованого додатка полягає у зчитуванні вхідних даних із пристроїв вводу-виводу, виконанні обчислювального алгоритму для формування вихідних даних і запису цих вихідних даних у пристрої вводу-виводу з повторенням цієї процедури через фіксовані інтервали часу.

У цьому розділі термін "*процес*" використовується для позначення програми, яку виконує процесор. Термін "*потік*" позначає потік виконання в межах процесу, яких може бути кілька. У наступному розділі ці теми розглядаються більш докладно.

Процеси і потоки

Багато (але не всі) операційні системи підтримують концепцію багатопотокового виконання. **Потік** – це послідовність програмних інструкцій, яка виконується логічно ізольовано від інших потоків. Операційна система, що працює на одноядерному процесорі, створює ілюзію одночасного виконання кількох потоків за рахунок поділу за часом.

При поділі за часом планувальник операційної системи надає кожному готовому до запуску потоку період часу для виконання. Коли інтервал виконання потоку закінчується, планувальник перериває запущений потік і продовжує виконання наступного потоку зі своєї черги. Таким чином, планувальник дає кожному потоку трохи часу для виконання, перш ніж повернутися до початку списку і почати цикл заново.

В операційних системах, здатних підтримувати кілька виконуваних програм одночасно, термін "процес" відноситься до запущеного екземпляра комп'ютерної програми. Для кожного процесу система виділяє ресурси, такі як пам'ять і місце в черзі готових до виконання потоків планувальника.

Під час запуску на виконання процес складається з одного потоку. У міру виконання процес може створювати більше потоків.

Програмісти створюють багатопотокові додатки з різних причин, включно з такими.

- Один потік може виконувати введення-виведення, тоді як інший потік виконує основний алгоритм програми. Наприклад, основний потік може періодично оновлювати виведену на користувацький дисплей інформацію, у той час як інший потік очікує користувацького введення з клавіатури, перебуваючи в заблокованому стані.
- Додатки зі значними вимогами до обчислювальних ресурсів можуть використовувати переваги багатопроцесорних і багатоядерних комп'ютерних архітектур, розділяючи великі обчислювальні завдання на групи дрібніших

завдань, які можна виконувати паралельно. Виконуючи кожне з цих невеликих завдань в окремому потоці, програми дають планувальнику можливість призначати різні потоки для одночасного виконання на кількох ядрах.

Протягом свого життєвого циклу процес проходить через низку станів. Ось деякі стани процесу, що призначаються операційними системами.

- **Ініціалізація:** під час першого запуску процесу (можливо, подвійним клацанням користувача по значку на робочому столі) операційна система починає завантажувати код програми в пам'ять і призначати системні ресурси для його використання.
- **Очікування:** після завершення ініціалізації процесу він готовий до запуску. На цьому етапі його потік поміщається в чергу готових до виконання потоків планувальника. Процес залишається в стані очікування доти, доки планувальник не дозволить йому почати виконання.
- **Виконання:** потік виконує програмні інструкції, що містяться в його розділі коду.
- **Блокування:** потік переходить у цей стан, коли запитує введення-виведення з пристрою, що призводить до призупинення виконання. Наприклад, блокування зазвичай викликає читання даних із файлу. У заблокованому стані потік очікує завершення обробки запиту драйвером пристрою. Щойно працюючий потік переходить у стан блокування, планувальник може переключитися на інший готовий до виконання потік, доки виконується операція введення-виведення першого потоку. Коли ця операція завершується, заблокований потік повертається в стан очікування в черзі планувальника і рано чи пізно переходить у стан виконання, після чого обробляє результати операції введення-виведення.

Час виконання для готових до запуску процесів розподіляє планувальник. Процес планувальника відповідає за надання часу виконання всім системним і користувацьким потокам.

Планувальник – це керована перериваннями процедура, що виконується в періодичні інтервали часу, а також у відповідь на дії потоків, такі як ініціювання операції введення-виведення. Під час ініціалізації операційної системи до обробника переривань планувальника підключається періодичний таймер і запускається таймер планувальника.

Поки кожен процес перебуває у стані ініціалізації, ядро додає до свого списку запущених процесів структуру даних, так звану **блоком керування процесом** (process control block, PCB). Цей блок містить інформацію, необхідну системі для обслуговування процесу та взаємодії з ним протягом усього його життєвого циклу, включно з відомостями про виділення пам'яті та про файл, що містить виконуваний

код процесу. У більшості операційних систем для ідентифікації процесу використовується ціле число, яке залишається єдиним протягом усього його життєвого циклу.

У Windows інструмент **Resource Monitor** (Монітор ресурсів) (цей інструмент можна запустити, набравши *Resource Monitor* (Монітор ресурсів) у полі пошуку Windows і клацнувши по результату з позначенням **Resource Monitor** (Монітор ресурсів)) відображає інформацію про запущені процеси, включно з **ідентифікатором процесу** (Process IDentifier, PID), призначеним кожному процесу. У Linux команда top відображає процеси, що споживають найбільшу кількість системних ресурсів, ідентифікуючи кожен із них за його ідентифікатором PID.

Планувальник зберігає інформацію, пов'язану з кожним потоком, у **блоці керування потоком** (thread control block, TCB). У кожного процесу є список пов'язаних TCB, який містить мінімум один запис. TCB зберігає інформацію, що стосується потоку, таку як **контекст процесора**. Це набір даних, які ядро використовує для відновлення виконання потоку, що очікує або заблокований. Цей набір складається з таких елементів:

- збережені регістри процесора;
- покажчик стека;
- регістр прапорів;
- покажчик інструкцій.

За аналогією з PID кожен потік має **ідентифікатор потоку** (thread identifier, TID), представлений цілим числом, який є унікальним протягом свого життєвого циклу.

Планувальник використовує один або кілька алгоритмів планування для рівномірного розподілу часу виконання між системними та користувацькими процесами. З перших днів існування обчислювальної техніки широко використовуються дві основні категорії алгоритмів планування потоків – не витісняючи та витісняючи:

- алгоритми, що не витісняючи, надають потоку повний контроль над виконанням, дозволяючи йому виконуватися доти, доки він не завершиться або поки він сам не передасть керування планувальнику, щоб інші потоки отримали можливість виконання;
- при використанні **алгоритмів витіснення** планувальник має право зупинити запущений потік і передати керування виконання іншому потоку без підтвердження з боку першого потоку.

Коли витіснений планувальник перемикає виконання з одного потоку на інший, він здійснює такі дії:

1. Відбувається переривання за таймером, яке змушує планувальник почати виконання, або запущений потік виконує дію, яка спричиняє його блокування, наприклад ініціює операцію введення-виведення.
2. Планувальник копіює регістри процесора потоку, що знімається, у контекстні поля блока керування цим потоком (TCB).
3. Планувальник переглядає свій список готових до виконання потоків і визначає, який потік слід перевести в стан виконання.
4. Планувальник завантажує контекст вхідного потоку в регістри процесора.
5. Планувальник відновлює виконання вхідного потоку, переходячи до інструкції, на яку вказує програмний лічильник потоку.

Планування потоків відбувається з високою частотою – це означає, що код, який визначає роботу планувальника, має бути максимально ефективним. Зокрема, збереження та вилучення контексту процесора займає деякий час, тому розробники операційних систем докладають усіх зусиль для оптимізації коду перемикання контексту планувальника.

У будь-який момент часу в черзі може бути безліч процесів, які претендують на виконання, тому планувальник повинен гарантувати, що критичні системні процеси будуть виконуватися з необхідними інтервалами. Водночас, з погляду користувача, додатки повинні демонструвати досить швидку реакцію на дані, що вводяться користувачем, забезпечуючи при цьому прийнятний рівень продуктивності при виконанні тривалих обчислень.

Для ефективного керування цими суперечливими вимогами протягом багатьох років було розроблено різні алгоритми. Основною особливістю більшості алгоритмів планування потоків є використання пріоритетів процесів. У наступному розділі представлено кілька алгоритмів планування потоків на основі пріоритетів.

Алгоритми планування та пріоритет процесу

Операційні системи, що підтримують кілька процесів, зазвичай надають механізм пріоритизації, завдяки якому найважливіші системні функції отримують достатній час опрацювання, навіть коли система працює з великим навантаженням, і водночас виділяється достатній час для виконання користувацьких процесів із нижчим пріоритетом. Для різних типів операційних систем було розроблено кілька алгоритмів, що переслідують різні цілі в галузі продуктивності. Ось деякі алгоритми планування, які були популярні протягом багатьох років, починаючи з найпростіших.

- **Першим прийшов, перший обслужений** (first come, first served, FCFS). Цей алгоритм, що не витісняє, був поширений у застарілих операційних системах із пакетним опрацюванням. В алгоритмі планування FCFS кожному процесу надається контроль над виконанням, і він зберігає цей контроль доти, доки

виконання не буде завершено. Пріоритизація процесів відсутня, і час завершення будь-якого процесу залежить від часу виконання процесів, що передують йому у вхідній черзі.

- **Кооперативна багатопотоковість**. У ранніх версіях Windows і macOS використовували невитісняючу архітектуру багатопотоковості, яка покладалася на припущення, що кожен потік сам передає керування операційній системі через досить часті проміжки часу. При цьому від розробників додатків були потрібні значні зусилля, щоб один додаток не позбавляв інші додатки можливості виконання через нездатність передати керування через відповідні проміжки часу. Щоразу, коли операційна система отримувала керування, вона вибирала наступний потік для виконання з пріоритетного списку готових до виконання потоків.

- **Циклічне планування**. Циклічний планувальник, що витісняє, веде список готових до виконання потоків і надає кожному з них інтервал виконання по черзі, повертаючись на початок списку при досягненні його кінця. Цей підхід фактично встановлює рівні пріоритети для всіх процесів, надаючи кожному з них можливість виконання протягом певних періодів часу з інтервалами, що залежать від кількості процесів у списку планувальника.

- **Планування з витісненням і фіксованим пріоритетом**. У цьому алгоритмі кожному потоку присвоюється пріоритет, що вказує на важливість передання цьому потоку керування виконанням, коли він перебуває в стані очікування. Коли потік, що має вищий пріоритет, ніж поточний запущений потік, переходить у стан очікування, планувальник негайно зупиняє запущений потік і передає керування вхідному потоку. Планувальник веде список очікуючих процесів у порядку пріоритету, поміщаючи потоки з найвищим пріоритетом на початок черги. Застосування цього алгоритму може призвести до того, що потоки з нижчим пріоритетом не зможуть одержати будь-який час для виконання, якщо потоки з вищим пріоритетом монополізують доступний час виконання.

- **Частотно-монотонне планування** (rate-monotonic scheduling, RMS) – це алгоритм витіснення з фіксованим пріоритетом, який зазвичай використовують у системах реального часу з жорсткими термінами завершення (жорсткий термін завершення – це термін, який не можна пропустити). При використанні цього алгоритму потокам, які виконуються частіше, призначаються вищі пріоритети. Доки виконуються кілька критеріїв (інтервал виконання потоку дорівнює терміну завершення, взаємодії між потоками, що спричиняють затримку, неможливі, а час перемикання контексту незначний), і якщо максимально можливий час виконання кожного потоку менший за математично розраховану межу, терміни завершення гарантовано будуть дотримані.

- **Справедливе планування** – це планування, спрямоване на максимальне використання процесорного часу з гарантією того, що кожному користувачеві надається рівна кількість часу виконання. Замість призначення потокам числових пріоритетів фактичний пріоритет кожного потоку визначається кількістю часу виконання, який він спожив. У міру того як потік використовує все більше і більше процесорного часу, його пріоритет знижується, надаючи іншим потокам більше можливостей для запуску. Перевага такого підходу полягає в тому, що для інтерактивних користувачів, які не споживають значного часу виконання, швидкодія системи підвищується. Ядро Linux використовує алгоритм справедливого планування як планувальник за замовчуванням.
- **Багаторівнева черга зі зворотним зв'язком**. Цей алгоритм використовує кілька черг, кожна з яких має різний рівень пріоритету. Нові потоки додаються в кінець черги з найвищим пріоритетом. У кожному інтервалі планування планувальник надає можливість виконання потоку, що стоїть на початку черги з високим пріоритетом, і видаляє цей потік із черги, наближаючи час виконання інших потоків. З часом новостворений потік отримує можливість виконання. Якщо потік споживає весь наданий йому час виконання, його витісняють після завершення свого інтервалу і додають у кінець наступної черги з нижчим пріоритетом. Планувальник Windows являє собою багаторівневу чергу зі зворотним зв'язком.

Системний процес простою містить потік, який виконується за відсутності будь-якого потоку, призначеного користувачеві або системі, що перебуває у стані очікування. Процес простою може складатися всього з однієї інструкції процесора, що задає нескінченний цикл. Деякі операційні системи в періоди простою замість виконання циклу простою переводять систему в режим енергозбереження.

Частка процесорного часу, споживаного запущеними процесами, обчислюється шляхом визначення частки часу, протягом якого система виконувала активний потік у період вимірювання. На рис. 5.1 показано Resource Monitor (Монітор ресурсів) Windows, що демонструє запущені процеси, які споживають найбільшу в середньому частку процесорного часу.

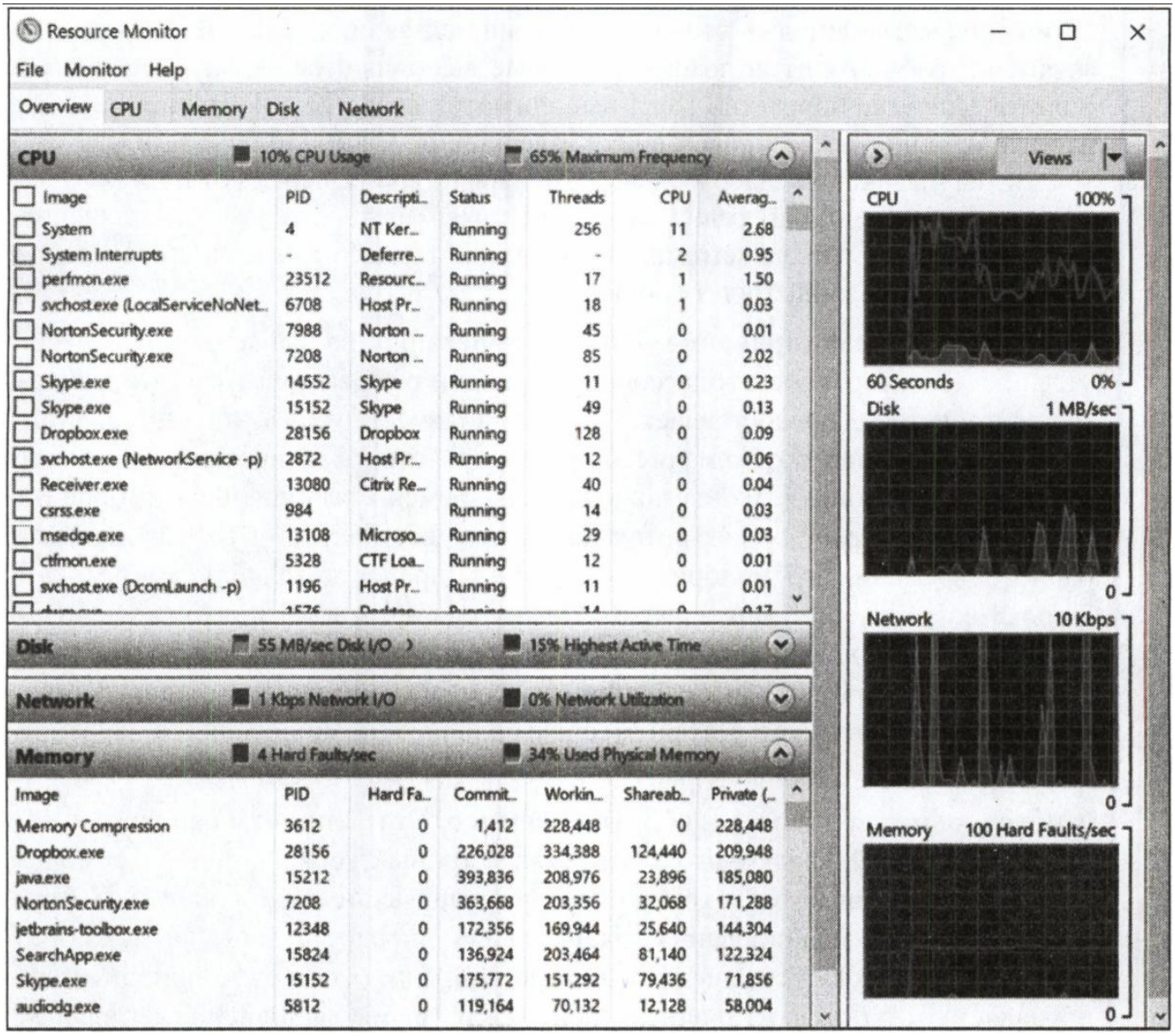


Рис 5.1. *Екран відомостей про процеси в Resource monitor Windows*

На знімку екрана показано, що в стовпчику **PID** (ІД процесу) відображаються числові ідентифікатори процесів, а в стовпчику **Threads** (Потоки) – кількість потоків у процесі. Усі процеси, показані на цьому екрані, перебувають у стані **Running** (Виконується).

На рис. 5.2 показано результат виконання команди *top* у системі Linux.

У верхній частині екрана міститься зведена інформація, включно з кількістю процесів, званих тут завданнями (*Tasks*), у кожному з можливих станів.

Кожен рядок у нижній частині екрана представляє інформацію про один запущений процес. Як і в Windows, у стовпчику *PID* показано ідентифікатори процесів. Стан кожного процесу відображено в стовпчику *S* з такими можливими значеннями:

- R – готовий до запуску: процес або запущений, або перебуває в черзі готових до запуску потоків;
- S – очікування: призупинено на час блокування; очікування завершення події;

- T – зупинений у відповідь на команду керування завданням (ця дія виконується при натисканні комбінації клавіш <Ctrl>+<Z>);
- Z – "зомбі": стан, що виникає, коли дочірній процес, що належить до іншого процесу, завершується, але система продовжує виводити інформацію про цей дочірній процес до завершення батьківського процесу.

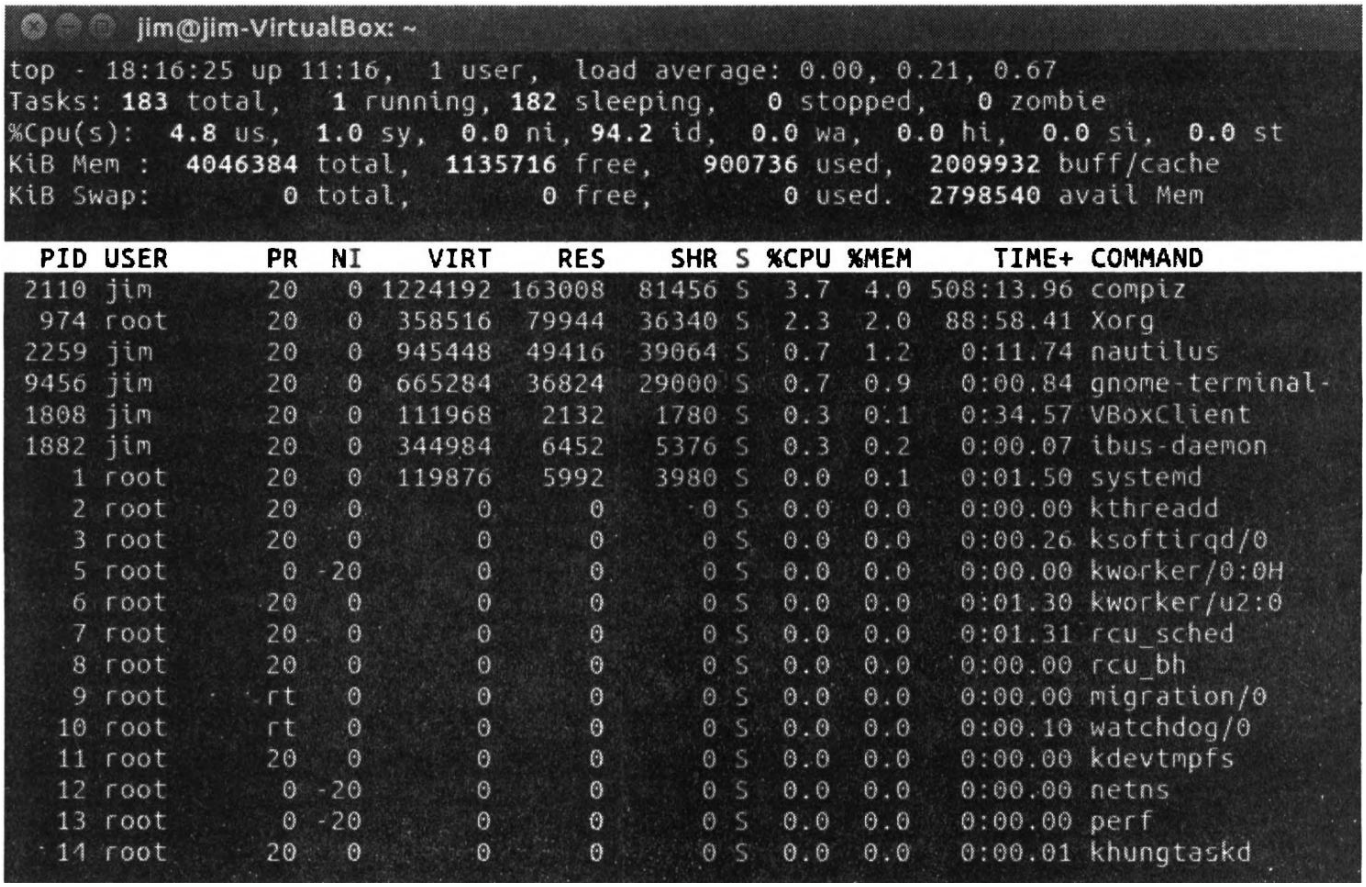


Рис 5.2. *Екран з інформацією про процеси Linux, виведений командою top*

У стовпчику *PR* відображається пріоритет планування процесу. Менші числа означають більш високі пріоритети.

До цього моменту ми говорили про процесор комп'ютера як про одиничну сутність. У більшості сучасних ПК інтегральна схема процесора містить кілька ядер, кожне з яких реалізує функції повноцінного незалежного процесора, включно з блоком керування, набором регістрів і АЛУ. У наступному розділі обговорюються атрибути систем, що містять кілька процесорних блоків.

Багатопроцесорність

Багатопроцесорний комп'ютер містить два процесори або більше, які одночасно виконують послідовності інструкцій. Процесори в такій системі зазвичай спільно використовують системні ресурси, наприклад основну пам'ять і периферійні пристрої. Процесори в багатопроцесорній системі можуть мати однакову архітектуру або окремі процесори можуть мати відмінну від інших архітектуру для підтримки унікальних

системних вимог. Системи, у яких усі процесори розглядаються як рівноправні, називають системами із **симетричною багатопроцесорною архітектурою**. Пристрої, що містять кілька процесорів у складі одного пакета інтегральних схем, називаються **багатоядерними процесорами**.

На рівні планувальника операційної системи симетричне багатопроцесорне середовище просто надає більше процесорів для використання в плануванні потоків. У таких системах планувальник розглядає додаткові процесори як ресурси під час призначення потоків для виконання.

У добре спроектованій симетричній багатопроцесорній системі пропускна спроможність може наблизитися до ідеального сценарію лінійного масштабування (пропорційно кількості доступних процесорних ядер) за умови, що конкуренція за спільні ресурси мінімальна. Наприклад, якщо кілька потоків на окремих ядрах спробують одночасно отримати масовий доступ до основної пам'яті, неминуче відбудеться зниження продуктивності, оскільки система розподіляє доступ до ресурсів між потоками, що конкурують. Багатоканальний інтерфейс динамічної пам'яті DRAM може підвищити загальну продуктивність системи в цьому сценарії.

Симетрична багатопроцесорна система є прикладом архітектури **з безліччю потоків інструкцій і безліччю потоків даних** (multiple instruction, multiple data, MIMD). MIMD – це конфігурація паралельної обробки, у якій кожне ядро процесора виконує незалежну послідовність інструкцій на власному наборі даних. Конфігурація паралельного опрацювання **з одним потоком інструкцій і безліччю потоків даних** (single instruction, multiple data, SIMD), для порівняння, одночасно виконує одну й ту саму інструкцію над кількома елементами даних.

Сучасні процесори реалізують інструкції SIMD для виконання паралельного опрацювання великих наборів даних, таких як графічні зображення і послідовності аудіоданих. У ПК поточного покоління використання багатоядерних процесорів забезпечує паралелізм виконання MIMD, а спеціалізовані інструкції всередині процесорів гарантують певний ступінь паралелізму виконання SIMD. Оброблення SIMD буде обговорюватися далі в *розділі 8*.

Тактова частота процесора зросла з 4,77 МГц в оригінальному IBM PC до більш ніж 4 ГГц у сучасних процесорах, що майже в тисячу разів більше. У майбутньому збільшення тактової частоти, ймовірно, буде менш динамічним, оскільки на цьому шляху перед нами постають фундаментальні фізичні обмеження. Для того щоб компенсувати обмежений приріст продуктивності за рахунок збільшення тактової частоти, процесорна галузь звернула особливу увагу на різні форми паралелізму виконання в персональних комп'ютерах та інтелектуальних пристроях. У майбутньому, імовірно, тенденція зростання паралелізму продовжиться, оскільки системи об'єднуватимуть десятки, потім сотні і в кінцевому рахунку тисячі

процесорних ядер, які паралельно працюватимуть на ПК, смартфонах та інших цифрових пристроях.

Резюме

Цей розділ розпочався з огляду драйверів пристроїв, включно з докладними відомостями про послідовності інструкцій, які використовуються кодом драйвера для читання і запису даних у простий пристрій введення-виведення: паралельний порт ПК. Потім ми обговорили системи – застарілу BIOS і нову UEFI, які надають код, який виконують одразу після ввімкнення ПК, що забезпечує тестування та ініціалізацію пристрою, а також завантаження операційної системи. Ми побачили, як процес довіреного завантаження допомагає гарантувати, що під час запуску системи дозволяється виконання тільки авторизованого коду, який не зазнав змін.

Потім ми перейшли до опису деяких фундаментальних елементів операційних систем, включно з процесами, потоками і планувальником. Було представлено різні алгоритми планування, що знайшли застосування в комп'ютерах минулого і в сучасних системах. Ми розглянули вихідні дані інструментів, реалізованих в операційних системах Linux і Windows, які надають інформацію про запущені процеси.

Розділ завершився обговоренням багатопроцесорного опрацювання та його впливу на продуктивність сучасних комп'ютерних систем, а також наслідків упровадження паралельного опрацювання в архітектурах MIMD і SIMD для майбутнього обчислювальної техніки.

У наступному розділі буде представлено різні галузі, де потрібні спеціалізовані обчислення, і їхні унікальні вимоги до обчислень у реальному часі, цифрової обробки сигналів і роботи **графічного процесора (GPU)**.

Вправи

1. Перезавантажте комп'ютер і увійдіть у розділ налаштувань BIOS або UEFI. Вивчіть кожне з доступних у цьому розділі меню. Яку систему використовує ваш комп'ютер, BIOS або UEFI? Чи підтримує ваша материнська плата розгін процесора? Коли ви закінчите, обов'язково виберіть варіант виходу без збереження змін, якщо тільки ви не маєте повної впевненості в тому, що хочете внести зміни.
2. Виконайте відповідну команду на комп'ютері, щоб відобразити інформацію про поточні запущені процеси. Який ідентифікатор Process ID (PID) має процес, який ви використовуєте для виконання цієї команди?

6

Спеціалізовані обчислення

Більшість користувачів комп'ютерів мають щонайменше загальне уявлення про ключові характеристики ПК та інтелектуальних цифрових пристроїв, пов'язані з продуктивністю, такі як швидкість процесора та обсяг оперативної пам'яті (random access memory, RAM). Цей розділ досліджує вимоги до продуктивності, властиві областям обчислень, які, як правило, менш очевидні для більшості користувачів, включно із системами реального часу, цифровим обробленням сигналів і обробленням даних у графічних процесорах (graphics processing unit, GPU).

Ми вивчимо унікальні обчислювальні функції, пов'язані з кожною з цих областей, і розглянемо приклади сучасних пристроїв, що реалізують ці функції.

Після ознайомлення з даним розділом ви знатимете, у яких галузях потрібні обчислення в реальному часі, і розбиратиметеся в основних концепціях цифрової обробки сигналів у контексті її широкого застосування в бездротовому зв'язку. Ви також ознайомитеся з базовою архітектурою сучасних графічних гіроцесорів і з деякими сучасними реалізаціями компонентів у галузях обчислень, що обговорюються в цьому розділі.

У цьому розділі розглядаються такі теми:

- обчислення в реальному часі;
- цифрове оброблення сигналів;
- обробка даних у графічних процесорах;
- приклади спеціалізованих архітектур.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою https://github.com/PacktPublishing/Modern-Computer-Architecture-and-OrganizationSecond-Edition.

Обчислення в реальному часі

У попередньому розділі було представлено короткий вступ до деяких вимог до обчислень у реальному часі з погляду здатності системи реагувати на зміни вхідних даних. Ці вимоги встановлюються у вигляді граничних термінів, що обмежують час, який може знадобитися системі для формування вихідних даних у відповідь на зміну вхідних даних. У цьому розділі ці тимчасові обмеження розглядаються детальніше, крім того, тут представлено деякі функції, що реалізуються в обчислювальних системах реального часу для дотримання часових вимог.

Обчислювальні системи реального часу можна розділити на системи з м'якими і жорсткими гарантіями реагування на зміни вхідних даних. Вважається, що **система м'якого реального часу** працює прийнятно, якщо вона виконує вимоги щодо очікуваного часу відгуку більшу частину часу, але не обов'язково весь час. Прикладом додатка, що працює в реальному часі, є відображення годинника на мобільному телефоні. Під час відкриття вікна з відображенням годинника деякі реалізації цього додатка на мить відображають час, що був показаний під час останнього перегляду годинника, після чого показання одразу ж оновлюються до правильного поточного часу. Користувачі, безумовно, хотіли б, щоб годинник показував правильний час щоразу, коли його відображають, але короткочасні збої, подібні до цього, зазвичай не розглядаються як суттєві проблеми.

З іншого боку, **система жорсткого реального часу** вважається непрацездатною, якщо вона коли-небудь пропускає будь-який зі встановлених термінів відгуку. Критично важливі системи безпеки, як-от контролери подушок безпеки в автомобілях і системи керування польотом цивільного авіалайнера, мають жорсткі вимоги до роботи в реальному часі. Розробники таких систем дуже серйозно ставляться до вимог щодо термінів реагування і докладають значних зусиль для того, щоб процесор реального часу задовольняв цим вимогам за будь-яких можливих умов експлуатації.

Потік керування простою системою реального часу показано на рис. 6.1.

На рис. 6.1 представлено обчислювальну систему реального часу, що використовує апаратний інтервальний таймер для керування часовою послідовністю своєї роботи.

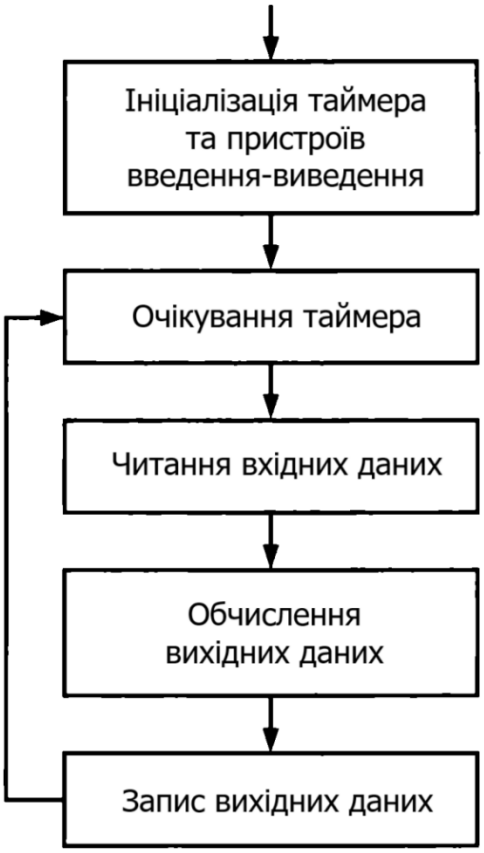


Рис 6.1. *Потік керування системою реального часу*

Інтервальний таймер зі зворотним відліком виконує повторюваний цикл таких дій:

1. Завантаження в регістр лічильника заздалегідь заданого числового значення.
2. Покрокове зменшення значення лічильника з фіксованою тактовою частотою.
3. Коли значення лічильника досягне нуля: формування події, наприклад встановлення біта в регістрі або виклик переривання процесора.
4. Повернення до кроку I.

Інтервальний таймер генерує періодичну послідовність подій із точністю, що залежить від характеристик системного годинника, для керування яким часто використовують кварцовий кристал. Очікуючи події таймера на початку кожного циклу, система, показана на рис. 6.1, починає кожен прохід виконання через фіксовані, майже рівні проміжки часу.

Для того щоб задовольнити вимоги жорсткого реального часу, час виконання коду всередині циклу (коду, який міститься в блоках "Читання вхідних даних", "Обчислення вихідних даних" і "Запис вихідних даних" на рис. 6.1) завжди має бути меншим, ніж інтервал таймера. Передбачливі розробники систем уважно стежать за тим, щоб жоден шлях через код не міг довести час виконання до межі жорсткого реального часу. Консервативне правило проектування системи може полягати в тому, що найдовший шлях виконання коду всередині циклу не повинен займати більше 50% інтервалу таймера.

Практичні системи реального часу, побудовані в такій конфігурації, можуть бути засновані на 8-, 16- або 32-розрядному процесорі, який працює на тактовій частоті в діапазоні від десятків до сотень мегагерц. Таймер, який використовується в основному циклі таких систем, генерує події з обраною розробником частотою, як правило, у діапазоні від 10 до 1000 Гц.

У менш складних системах код, представлений блоками на рис. 6.1, часто виконується безпосередньо на апаратних засобах процесора без задіяння проміжних програмних рівнів. Ця конфігурація не передбачає використання операційної системи реального часу, описаної в *розділі 5*. З іншого боку, складний додаток реального часу цілком ймовірно матиме ширші потреби, ніж ті, що можна реалізувати за допомогою цієї спрощеної архітектури, і це робить привабливим використання операційної системи реального часу.

Операційні системи реального часу

Операційна система реального часу (ОСРЧ) має низку особливостей, зовні схожих з операційними системами загального призначення, розглянутими в *розділі 5*. Однак архітектура ОСРЧ значно відрізняється від операційних систем загального призначення в тому, що всі аспекти ОСРЧ – від внутрішніх структур ядра до драйверів пристроїв і системних сервісів – зосереджені на задоволенні вимог жорсткого реального часу.

У більшості ОСРЧ використовується принцип витісняючої багатопоточності, який часто називають у літературі з ОСРЧ **багатозадачністю**. Терміни "завдання" і "потік" у контексті ОСРЧ є до певної міри синонімами, тому для послідовності викладу ми продовжуватимемо використовувати термін "потік" для позначення завдань ОСРЧ.

Архітектури ОСРЧ на нижньому кінці шкали складності зазвичай підтримують багатопоточність у контексті одного прикладного процесу. Ці простіші ОСРЧ підтримують пріоритизацію потоків, але часто не мають функцій захисту пам'яті.

Складніші архітектури ОСРЧ надають такі функції операційної системи, як захист пам'яті, на додаток до витісняючої багатопотоковості з пріоритизацією. У цих ОСРЧ кілька процесів можуть одночасно перебувати в стані Running (Виконується), при цьому кожен із процесів може містити кілька потоків. У системах із захистом пам'яті доступ до пам'яті ядра з боку потоків застосунків заборонений, і застосунки не можуть проникати в області пам'яті один одного.

Середовища ОСРЧ, від найнижчого до найвищого рівня складності, надають кілька структур даних і методів взаємодії, спрямованих на забезпечення ефективного обміну даними між потоками, а також на підтримку контрольованого доступу до спільних ресурсів. Ці можливості часто доступні, і не тільки в ОСРЧ. Ось кілька прикладів таких можливостей.

- **Взаємне виключення**, або **м'ютекс** {mutex, від англ. mutual exclusion), – це механізм, що дозволяє потоку вимагати доступ до спільного ресурсу, не блокуючи виконання інших потоків. У своїй найпростішій формі взаємне виключення реалізується за допомогою доступної всім потокам змінної, що має значення 0, коли ресурс вільний, та 1, коли ресурс зайнятий. Потік, якому потрібен ресурс, зчитує поточне значення такої змінної, і якщо воно дорівнює 0, привласнює їй значення 1, після чого виконує операцію з використанням цього ресурсу. Після завершення операції потік повертає цій змінній значення 0. Водночас використання взаємних виключень супроводжується низкою таких потенційних проблем.

  - **Витіснення потоків**. Припустімо, потік зчитує змінну взаємного винятку і виявляє, що вона дорівнює 0. Оскільки планувальник здатний перервати потік, що виконується, у будь-який час, цей потік може бути перервано до того, як він встигне встановити для змінної значення 1. Потім поновлюється виконання іншого потоку, який бере під контроль той самий ресурс, оскільки бачить, що змінна все ще дорівнює 0. Коли поновлюється виконання першого потоку, він закінчує встановлення для змінної значення 1 (хоча до цього моменту вона вже дорівнює 1). У цей момент обидва потоки помилково вважають, що в них є винятковий доступ до цього ресурсу, що найімовірніше, призведе до серйозних проблем, якщо обидва потоки спробують використовувати цей ресурс.

    Для того щоб запобігти такому сценарію, у багатьох процесорах реалізують деяку форму **інструкції "перевірити-встановити"**. Інструкція "перевірити-встановити" зчитує значення з комірки пам'яті та встановлює для цієї комірки значення 1 за одну безперервну дію (також звану **неподільною**). В архітектурі x86 цю операцію виконує інструкція *BTS* (перевірка і встановлення біта). В архітектурах процесорів, де відсутня інструкція типу "перевірити-встановити" (наприклад, 6502), ризик витіснення можна усунути шляхом вимкнення переривань перед перевіркою стану змінної взаємного виключення і наступного ввімкнення переривань після присвоєння цій змінній значення 1. Недоліком цього підходу є зниження швидкості реагування в реальному часі в період вимкнення переривань.

  - **Інверсія пріоритетів** відбувається, коли потік із вищим пріоритетом намагається отримати доступ до ресурсу, тоді як відповідна змінна взаємного винятку утримується під контролем потоку з нижчим пріоритетом. У такій ситуації ОСРЧ зазвичай переводять більш пріоритетний потік у заблокований стан, дозволяючи низькопріоритетному потоку завершити свою роботу і звільнити тимчасову.

    Проблема з інверсією пріоритетів виникає, коли починається виконання потоку з пріоритетом, що знаходиться між рівнями пріоритетів перших двох

потоків. Під час роботи цього середньопріоритетного потоку виконання низькопріоритетного потоку зупинено, і він не може звільнити змінну взаємного виключення. Високопріоритетний потік тепер повинен чекати, поки закінчиться виконання середньопріоритетного потоку, що фактично порушує всю схему пріоритетів потоків. Це може призвести до того, що високопріоритетний потік не зможе вкластися в заданий термін.

Одним із методів запобігання інверсії пріоритетів є **наслідування пріоритетів**. В ОСРЧ, де реалізовано успадкування пріоритетів, щоразу, коли потік із вищим пріоритетом (*hi_thread*) запитує доступ до змінної взаємного винятку, що утримується потоком з нижчим пріоритетом (*lo_thread*), пріоритет потоку *lo_thread* тимчасово підвищується до рівня *hi_thread*. Це виключає будь-яку можливість того, що потік із середнім пріоритетом затримає завершення (спочатку) нижчого пріоритетного потоку *lo_thread*. Коли потік *lo_thread* звільняє змінну взаємного виключення, ОСРЧ відновлює початковий пріоритет цього потоку.

- **Взаємне блокування** може виникнути, коли кілька потоків намагаються заблокувати кілька змінних взаємного виключення. Якщо потоки *thread1* і *thread2* потребують контролю над змінними взаємного винятку *mutex1* і *mutex2*, може виникнути ситуація, у якій потік *thread1* блокує змінну *mutex1* і намагається заблокувати змінну *mutex2*, у той час як потік *thread2* уже заблокував змінну *mutex2* і намагається заблокувати змінну *mutex1*. У такій ситуації жодне з цих завдань не може виконуватися, звідси і термін "взаємне блокування". Деякі реалізації ОСРЧ перевіряють приналежність змінних взаємного виключення під час спроби їх заблокувати та повідомляють про помилку в ситуації взаємного блокування. У простіших ОСРЧ розробник системи має переконатися, що виникнення взаємного блокування неможливе.

- **Семафор** – це узагальнення змінної взаємного виключення. Розрізняють семафори двох типів: двійковий і підраховуваний.
  - **Двійковий семафор** схожий на взаємне виключення, з тією різницею, що він скоріше призначений не для контролю над доступом до ресурсу, а для того, щоб одне завдання могло надіслати сигнал іншому завданню. Якщо потік *thread1* намагається взяти під контроль семафор *semaphore1*, коли він недоступний, *thread1* блокується доти, доки інший потік або процедура обслуговування переривань не передасть контроль над семафором *semaphore1*.
  - **Підраховуваний семафор** містить лічильник із верхньою межею. Підраховувані семафори використовуються для контролю доступу до кількох взаємозамінних ресурсів. Коли потік бере під контроль підраховуваний

семафор, значення лічильника збільшується, і завдання виконується. Коли лічильник досягає своєї межі, потік, що намагається взяти під контроль семафор, блокується доти, доки інший потік не віддасть контроль над цим семафором, зменшивши значення лічильника.

Розглянемо приклад системи, яка підтримує обмежену кількість одночасно відкритих файлів.

Для керування операціями відкриття і закриття файлів можна використовувати семафор, що підраховує. Якщо система підтримує до 10 відкритих файлів і потік намагається відкрити 11-й файл, підраховуваний семафор з межею 10 блокуватиме операцію відкриття файлу, допоки не буде закрито інший файл і не стане доступний його дескриптор.

- **Черга** (інша назва – **черга повідомлень**) – це односпрямований шлях зв'язку між процесами або потоками. Потік-відправник поміщає елементи даних у чергу, а потік-одержувач витягує ці елементи в тому самому порядку, у якому їх було надіслано. ОСРЧ синхронізує доступ між відправником і одержувачем, тому одержувач отримує тільки повні елементи даних. Черги зазвичай реалізуються з буфером зберігання фіксованого розміру. Буфер із часом заповнюється і блокує подальше додавання даних, якщо потік-відправник додає елементи даних швидше, ніж потік-одержувач їх витягує.

  Черги повідомлень ОСРЧ надають програмний інтерфейс, що дозволяє потоку одержувачу перевірити, чи містить черга дані. Багато реалізацій черг також підтримують використання семафора для подання сигналу заблокованому потоку одержувачу про появу даних.

- **Критична секція.** Звичайною є ситуація, за якої кільком потокам потрібен доступ до загальної структури даних. Під час використання спільних даних дуже важливо, щоб операції читання і запису, що виконуються різними потоками, не накладалися в часі одна на одну. Коли таке накладання відбувається, потік, що зчитує, може отримати суперечливу інформацію, якщо він звертається до структури даних у той час, коли інший потік виконує її оновлення. Механізми взаємного виключення і семафора надають можливості для керування доступом до загальних структур даних. Використання критичної секції – це альтернативний підхід, який ізолює код, що звертається до загальної структури даних, і дозволяє тільки одному потоку в будь-який момент часу виконувати цю послідовність.

  Простим методом реалізації критичної секції є вимкнення переривань безпосередньо перед входом до критичної секції та їхнє подальше увімкнення після виконання критичної секції. Це запобігає запуску планувальника і надає потоку, що звертається до структури даних, можливість одноосібного керування до виходу з критичної секції. Однак цей метод має недолік – він знижує

швидкість реакції в реальному часі, оскільки не дає змоги реагувати на переривання, зокрема виконувати функції планування потоків, поки переривання вимкнено.

Деякі ОСРЧ пропонують складнішу реалізацію методу критичної секції з використанням об'єктів даних критичної секції. Об'єкти критичної секції зазвичай надають можливість вибору: потік може або перейти в заблокований стан, поки критична секція не стане доступною, або перевірити, чи використовується критична секція, без блокування. Функція перевірки доступності критичної секції дає змогу потоку виконувати іншу роботу в очікуванні, поки критична секція звільниться.

Сьогодні в експлуатації перебуває набагато більше обчислювальних систем реального часу, ніж персональних комп'ютерів, які ми сприймаємо як *комп'ютери*. Комп'ютери загального призначення становлять менше 1% цифрових процесорів, що випускаються щороку. Найрізноманітніші пристрої – від дитячих іграшок до цифрових термометрів, телевізорів, автомобілів і космічних кораблів – містять щонайменше один, а найчастіше й десятки вбудованих процесорів, кожний з яких працює під керуванням ОСРЧ того чи іншого типу.

У цьому розділі було дано короткий вступ у деякі можливості обміну даними та керування ресурсами, поширені в різних реалізаціях ОСРЧ. У наступному розділі представлено архітектури, використовувані під час обробки цифрових виборів аналогових сигналів.

Цифрове оброблення сигналів

Процесор для цифрової обробки сигналів (digital signal processor, DSP) – це процесор, оптимізований для виконання обчислень над оцифрованими представленнями аналогових сигналів. Використовувані в реальному світі сигнали, такі як аудіо, відео, **радіочастотні** (РЧ) сигнали стільникових телефонів і сигнали радарів, є аналоговими за своєю природою. Це означає, що оброблювана інформація – це реакція електричного датчика на вхідну напругу, що безперервно змінюється. Перш ніж цифровий процес може почати працювати з аналоговим сигналом, напруга сигналу має бути перетворена в цифрову форму за допомогою **аналого-цифрового перетворювача** (АЦП). У наступному розділі описується робота АЦП і **цифро-аналогових перетворювачів** (ЦАП).

АЦП і ЦАП

АЦП вимірює аналогову вхідну напругу і видає на виході цифрове слово, що представляє величину вхідної напруги. ЦАП виконує операцію, зворотну АЦП, перетворюючи цифрове слово в аналогову напругу. У процесі перетворення АЦП часто використовують внутрішній ЦАП.

У застосовуваних на практиці ЦАП використовуються схеми різних архітектур. Це робиться, як правило, з метою отримання необхідного поєднання низької вартості, високої швидкості та високої точності. Однією з найпростіших архітектур ЦАП є матриця R-2R, показані на рис. 6.2 у конфігурації з 4-розрядним входом.

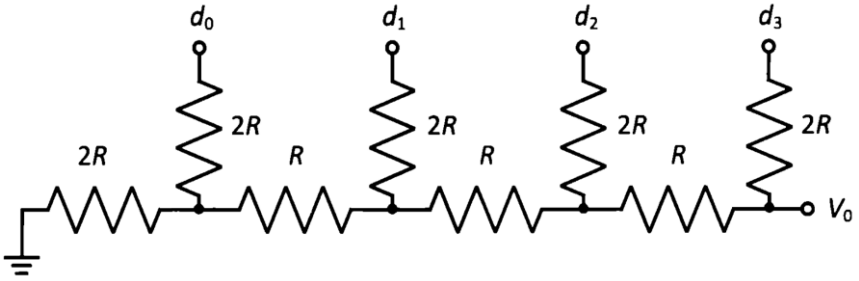


Рис 6.2. *ЦАП матричного типу R–2R*

Цей ЦАП використовує 4-розрядне слово даних на входах d_0-d_3 (де d є найбільш значущим бітом) для одержання аналогової напруги V_0. Якщо припустити, що стан кожного біта 4-розрядного слова d визначається напругою 0 В (для біта зі значенням 0) або 5 В (для біта зі значенням 1), вихідна напруга U дорівнюватиме $(d/2^4) * 5$ В, де d є значенням у діапазоні від 0 до 15. Вхідне слово зі значенням 0 дає на виході напругу 0 В, а вхідне слово зі значенням $15 - (15/16) * 5$ В = 4,6875 В. Проміжні значення слова d дають на виході рівномірно розподілені напруги з кроком $(1/16) * 5$ В = 0,3125 В.

АЦП може використовувати внутрішній ЦАП, подібний до описаного вище (хоча зазвичай із більшою розрядністю і, відповідно, вищою роздільною здатністю за напругою), разом зі схемою вибірки-зберігання для визначення цифрового еквівалента аналогової вхідної напруги. Оскільки аналоговий вхідний сигнал може змінюватися безперервно з плином часу, схеми АЦП зазвичай використовують схему **вибірки-зберігання** для підтримки постійної аналогової вхідної напруги в процесі перетворення. Схема вибірки-зберігання являє собою аналоговий пристрій із дискретним вхідним сигналом утримання. Коли вхід утримання неактивний, вихід схеми повторює вхідну напругу. Коли вхід утримання активний, схема вибірки-зберігання фіксує й утримує вихідну напругу на рівні вхідної напруги, яка була присутня в момент появи сигналу утримання.

При постійному сигналі на виході схеми вибірки-збереження АЦП використовує свій ЦАП для визначення цифрового еквівалента вхідної напруги. Для цього АЦП використовує **компаратор**, який являє собою схему, що порівнює дві аналогові напруги і видає дискретний вихідний сигнал, який вказує, яка з двох напруг вища. АЦП подає вихідну напругу схеми вибірки-зберігання на один вхід компаратора, а вихідну напругу ЦАП – на інший його вхід, як показано на рис. 6.3, де розмір вхідного слова ЦАП дорівнює n бітам.

Завдання АЦП полягає в знаходженні вхідного слова ЦАП, яке призводить до зміни стану компаратора. Простий спосіб зробити це – вести відлік від нуля, записуючи

кожне числове значення на вхід ЦАП і спостерігаючи за входом компаратора, щоб зрозуміти, чи змінився його стан. Вихідний сигнал ЦАП, який першим викличе зміну стану компаратора, є найменшою вихідною напругою ЦАП, що є більшою за вихідну напругу схеми вибірки-зберігання. Фактична зафіксована аналогова напруга знаходиться між цим значенням вихідної напруги ЦАП і її значенням, коли на вхід ЦАП подається слово даних на один відлік менше. АЦП такої конфігурації називають **АЦП послідовного рахунку**.

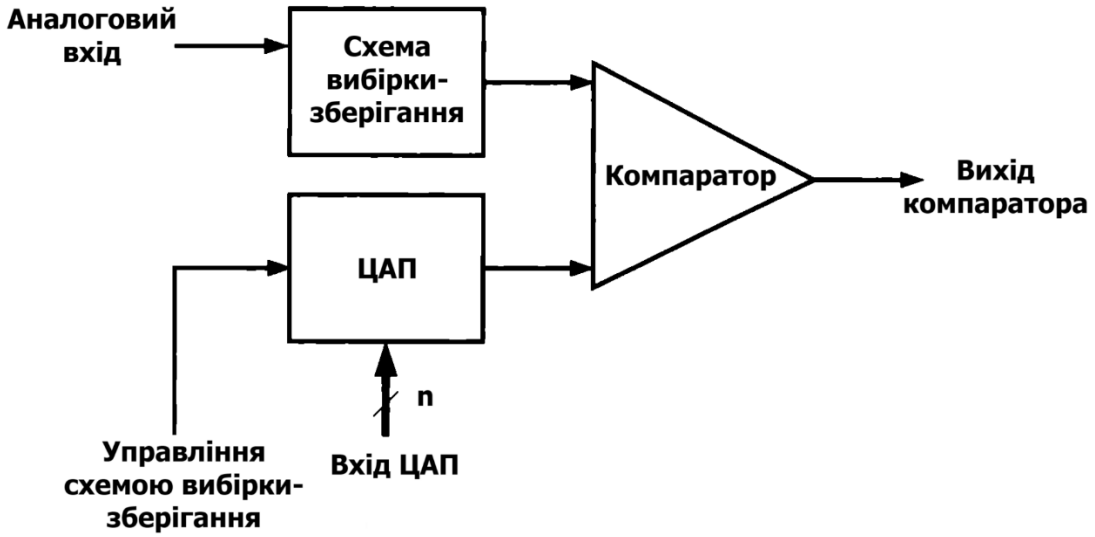


Рис 6.3. *Архітектура АЦП*

Незважаючи на просту концепцію, АЦП послідовного рахунку може бути досить повільним, особливо при великому розмірі слова. Більш швидкий метод полягає в послідовному порівнянні кожного біта в слові даних ЦАП, починаючи з найбільш значущого біта (d_3) Якщо почати зі слова даних 1000b для нашого прикладу з 4-розрядною схемою, то перше показання компаратора дасть змогу визначити місце розташування рівня аналогової вхідної напруги – вище або нижче середньої точки діапазону напруги ЦАП. У такий спосіб визначається значення біта d_3 показання АЦП: 0 або 1. Використовуючи відоме значення d_3, встановлюємо d_2 рівним 1, щоб визначити, у межах якої чверті повного діапазону перебуває вхідна напруга. Ця процедура повторюється для послідовного визначення кожного з бітів, що залишилися, і закінчується на молодшому значущому біті.

Такий метод перетворення в АЦП називається послідовним наближенням. АЦП послідовного наближення працює набагато швидше, ніж АЦП послідовного рахунку. У нашому прикладі максимально можлива кількість порівнянь зменшується з 16 до 4. У 12-розрядному АЦП послідовного наближення максимально можливе число порівнянь зменшується з 4096 до 12. У загальному випадку, використання методу послідовного наближення замість послідовного лічби зменшує максимальну кількість кроків для n-розрядного АЦП з 2^n до n. АЦП послідовного наближення мають роздільну здатність від 8 до 18 біт і максимальну частоту перетворення до кількох мегагерц.

АЦП і ЦАП характеризуються роздільною здатністю і максимальною швидкістю перетворення. Роздільна здатність АЦП або ЦАП визначається кількістю бітів у його слові даних. Максимальна швидкість перетворення визначає, наскільки швидко АЦП або ЦАП може видавати послідовні вхідні дані.

Під час обробки даних у реальному часі АЦП виконує послідовність вимірювань через періодичні проміжки часу, отримуючи вхідні дані для подальшої обробки. Вимоги до роздільної здатності слів даних і частоти дискретизації сильно різняться залежно від конкретного застосування DSP. Ось деякі приклади стандартних форматів оцифрованих аналогових даних.

- Цифровий звук для компакт-дисків дискретизується з частотою 44,1 кГц і 16 бітами на вибірку у двох каналах, що відповідають лівому і правому динамікам.
- Відеокамери вимірюють аналоговий сигнал інтенсивності світла, отриманий від кожного пікселя двовимірної матриці, і перетворюють отримані показники в цифрове слово розміром, як правило, 8 біт. Окремі, розташовані близько один до одного датчики з колірними фільтрами проводять виміри для червоного, зеленого і синього кольорів у кожному пікселі зображення. Повний набір даних для одного пікселя налічує 24 біти, що складаються з трьох 8-бітних значень кольору. Одне зображення може містити десятки мільйонів пікселів, а відеокамери зазвичай видають від 30 до 60 кадрів на секунду. Через те, що цифрове відео виробляє величезний обсяг даних, для зниження вимог до їхнього зберігання та передавання зазвичай використовують алгоритми стиснення.
- Мобільний телефон містить радіочастотний приймач, який перетворює прийнятий радіосигнал у частотний діапазон, що підходить для введення в АЦП. Типові параметри для АЦП мобільного телефону: 12-бітна роздільна здатність і частота дискретизації 50 МГц.
- Автомобільна радарна система відбирає радіочастотну енергію, відбиту від прилеглих перешкод, з роздільною здатністю 16 біт і частотою 5 МГц.

У наступному розділі ми розглянемо вимоги до обробки сигналів для послідовностей даних, захоплених АЦП.

Особливості апаратних засобів DSP

Процесори цифрового оброблення сигналів (DSP) оптимізовані для виконання алгоритмів оброблення цифрових виборів аналогової інформації. **Скалярний добуток** – це фундаментальна операція, яка використовується в багатьох алгоритмах, що виконуються DSP. Якщо $\vec{a}$ і $\vec{b}$ являють собою два вектори однакової довжини (**вектор** – це одновимірний масив числових значень), то їхній скалярний добуток обчислюється шляхом множення кожного елемента $\vec{a}$ на відповідний елемент $\vec{b}$ і

підсумовування отриманих добутків. З математичної точки зору, якщо довжина кожного вектору дорівнює n (з індексами від 0 до $n-1$), то скалярний добуток векторів дорівнює:

$$\vec{a} * \vec{b} = \sum_{i=0}^{n-1} a_i b_i = a_0 b_0 + a_1 b_1 + a_2 b_2 + \ldots + a_{n-1} b_{n-1}.$$

Повторюваний характер обчислення скалярного добутку вказує природний шлях для оптимізації продуктивності в цифрових системах. Основна операція, що виконується під час обчислення скалярного добутку, називається **множенням із накопиченням** (multiply accumulate, MAC).

Одна операція MAC складається з перемноження двох чисел і додавання результату в акумулятор, який був ініціалізований шляхом скидання перед обчисленням скалярного добутку. Математична продуктивність чипів DSP зазвичай вимірюється в кількості операцій MAC на секунду. Багато архітектур DSP здатні виконати одну операцію MAC за один тактовий цикл виконання інструкції.

Для того щоб виконувати таку операцію в кожному такті, DSP не може виокремлювати окремі такти для читання інструкції MAC з пам'яті програм, читання кожного з векторів, що множаться, з пам'яті даних, обчислення добутку і додавання його в акумулятор. Усі ці дії необхідно виконати за один крок.

Архітектура фон Неймана, представлена в *розділі 1*, використовує єдину область пам'яті для програмних інструкцій і даних. Така конфігурація призводить до обмеження, відомого як **вузьке місце архітектури фон Неймана**, що виникає внаслідок необхідності передавати як інструкції програм, так і значення даних через єдиний інтерфейс "процесор – пам'ять".

Це обмеження можна усунути за допомогою архітектури, яка розділяє сховище програм і даних на дві окремі області пам'яті, кожна з яких має власний інтерфейс для обміну даними з процесором. У такій конфігурації, так званій **гарвардською архітектурою**, програмні інструкції та операції доступу до пам'яті даних можна виконувати паралельно, завдяки чому потрібна менша кількість тактів. Гарвардську архітектуру буде обговорено детальніше в *розділі 7*.

Процесор DSP з гарвардською архітектурою повинен виконати дві операції звернення до пам'яті даних для витягання елементів векторів $\vec{a}$ і $\vec{b}$, які потрібно перемножити за допомогою операції MAC. Зазвичай для цього потрібні два такти, що не дозволяє досягти цільового показника продуктивності в одну операцію MAC за один такт. **Модифікована гарвардська архітектура** підтримує використання пам'яті програм для зберігання значень даних на додаток до інструкцій. У багатьох додатках DSP значення одного з векторів (скажімо, вектору $\vec{a}$), що беруть участь в операції скалярного добутку, є постійними величинами, відомими на момент компіляції додатка. У модифікованій гарвардській архітектурі елементи вектору $\vec{a}$, що

зчитуються з АЦП як вхідні дані, можуть зберігатися в пам'яті програм, а елементи вектору $\vec{b}$ – в пам'яті даних.

Для виконання кожної операції MAC у цій архітектурі один елемент вектору $\vec{a}$ зчитується з пам'яті програм, один елемент вектора $\vec{b}$ зчитується з пам'яті даних, а результат, що підлягає накопиченню, зберігається у внутрішньому регістрі процесора. Якщо DSP містить кеш-пам'ять для програмних інструкцій, інструкцію MAC, що виконує кожний крок для обчислення скалярного добутку, буде витягнуто з кеша, щойно перша операція MAC зчитає її з пам'яті програм, що дасть змогу унеможливити подальші цикли доступу до пам'яті для введення цієї інструкції. Така конфігурація (модифікована гарвардська архітектура з кешуванням програмних інструкцій) дозволяє використовувати операції MAC, що виконуються за один цикл, для всіх ітерацій знаходження скалярного добутку після завершення першої операції MAC. Оскільки вектори, що беруть участь у реальних обчисленнях скалярного добутку, зазвичай містять сотні або навіть тисячі елементів, загальна продуктивність операцій обчислення скалярного добутку може досить близько підійти до ідеалу – однієї операції MAC за такт DSP.

Процесори DSP можна класифікувати за ознакою підтримки їхньою архітектурою операцій із фіксованою та плаваючою комою. Процесори DSP з фіксованою комою для виконання математичних операцій, таких як MAC, використовують цілі числа зі знаком або без знаку. Процесори DSP з фіксованою комою, як правило, дешевші, ніж процесори DSP з плаваючою комою. Однак математика з фіксованою комою може призвести до проблем з числами, таких як переповнення, яке може проявлятися в перевищенні діапазону акумулятора результатів скалярних добутків.

Для того щоб зменшити ймовірність переповнення, для підтримки обчислень скалярного добутку довгих векторів у DSP часто використовують акумулятор розширеного діапазону, наприклад шириною 40 біт у 32-розрядній архітектурі. Через побоювання з приводу переповнення і пов'язаних з цим проблем програмування DSP з фіксованою комою вимагає додаткових зусиль для виключення ймовірності того, що ці ефекти призведуть до неприйнятного зниження продуктивності.

У DSP із плаваючою комою для внутрішніх обчислень часто використовують 32-розрядні числа. Якщо DSP отримує цілочисельне значення АЦП, усе подальше опрацювання виконується з використанням операцій із плаваючою комою. Завдяки використанню переваг операцій із плаваючою комою ймовірність виникнення таких проблем, як переповнення, значно знижується, що дає змогу скоротити цикли розроблення програмного забезпечення.

Використання обробки з плаваючою комою також підвищує достовірність результатів обчислень, що виражається в поліпшенні **відношення "сигнал/шум"** (signal-to-noise ratio, SNR) порівняно з еквівалентною реалізацією з фіксованою комою. Обчислення з фіксованою комою квантують результат кожної математичної операції на рівні

молодшого значущого біта цілого числа. Операції з плаваючою комою, як правило, дають точні результати кожної операції, при цьому їхня точність перебуває в межах незначної частки відповідного молодшого значущого біта числа з фіксованою комою.

Алгоритми обробки сигналів

Ґрунтуючись на нашому розумінні апаратних засобів DSP і операцій, які вони підтримують, далі ми розглянемо деякі приклади алгоритмів цифрового оброблення сигналів у реальних додатках.

Згортка

Згортка – це формальна математична операція нарівні зі складанням і множенням. На відміну від додавання і множення, які працюють з парами чисел, згортка працює з парами сигнальних векторів. У контексті DSP сигнал являє собою серію оцифрованих виборів змінного в часі вхідного сигналу, відібраних через рівні проміжки часу. Згортання – це найфундаментальніша операція в галузі DSP.

У багатьох практичних додатках одним із двох сигналів, що беруть участь в операції згортання, є фіксований вектор чисел, що зберігаються в пам'яті. Інший сигнал являє собою послідовність вибірок, отриманих у результаті вимірювань за допомогою АЦП.

Для того щоб реалізувати операцію згортання, під час отримання кожного вимірювання АЦП процесор DSP обчислює змінений вихідний сигнал, що є скалярним добутком фіксованого вектору даних (припустімо, довжина цього вектору дорівнює *n*) та найостанніших вхідних вибірок у кількості *n*, отриманих від АЦП. Для того щоб обчислити згортку цих векторів, щоразу під час отримання вибірки від АЦП процесор DSP має виконати *n* операцій MAC.

Фіксований вектор у цьому прикладі, що позначається $\vec{h}$, називається **імпульсною характеристикою**. Цифровий імпульс визначається як теоретично нескінченна послідовність відліків, у якій один відлік дорівнює 1, а всі попередні та наступні відліки дорівнюють 0. Використання цього вектору як вхідних даних для згортання з вектором $\vec{h}$ призводить до отримання вихідних даних, тотожних послідовності $\vec{h}$, оточеній попередніми і наступними нулями. Єдине значення 1 в імпульсній послідовності множить кожен елемент $\vec{h}$ у послідовних ітераціях, тоді як усі інші елементи $\vec{h}$ множаться на 0.

Конкретні значення, що містяться у векторі $\vec{h}$, визначають вплив операції згортання на послідовність вхідних даних. Цифрова фільтрація є одним із поширених застосувань згортання.

Цифрова фільтрація

Частотно-вибірковий фільтр являє собою схему або алгоритм, який приймає вхідний сигнал і передає його складові в потрібних частотних діапазонах на вихід без спотворень, водночас усуваючи або, принаймні, зменшуючи до прийнятного рівня його складові в частотних діапазонах за межами потрібних діапазонів.

Ми всі знайомі з регуляторами низьких і високих частот в аудіосистемах. Це приклади частотно-вибіркових фільтрів. Функція баса реалізує **фільтр нижніх частот** зі змінним посиленням. Це означає, що аудіосигнал фільтрується для вибору його низькочастотної частини, після чого цей відфільтрований сигнал подається на підсилювач, який змінює свою вихідну потужність залежно від положення регулятора баса. Секція високих частот реалізована аналогічним чином, з використанням **фільтра верхніх частот** для вибору більш високих частот у звуковому сигналі. Виходи цих підсилювачів об'єднуються для отримання сигналу, що посилається на динаміки.

Частотно-вибіркові фільтри можна реалізувати за допомогою аналогової технології або методів цифрової обробки сигналів. Прості аналогові фільтри дешеві й потребують лише кількох схемних компонентів. Однак робочі характеристики цих простих фільтрів залишають бажати кращого.

Одними з основних параметрів частотно-вибіркового фільтра є **загасання в смузі затримування і ширина перехідної смуги**. Загасання в смузі затримування показує, наскільки добре фільтр справляється з усуненням небажаних частот на виході. Як правило, фільтр не усуває небажані частоти повністю, але для практичних цілей ці частоти можна зменшити до настільки малого рівня, що вони перестають мати якесь значення.

Перехідна смуга фільтра – це діапазон частот між смугою пропускання і смугою затримування. **Смуга пропускання** – це діапазон частот, які мають бути пропущені через фільтр, а **смуга затримування** – це діапазон частот, які мають бути заблоковані фільтром. Неможливо забезпечити ідеально чітку межу між смугою пропускання і смугою затримування. Між цими смугами є деяка перехідна смуга, і щоб зробити її якомога вужчою, потрібен складніший фільтр, ніж фільтр із ширшою перехідною смугою.

Цифровий частотно-вибірковий фільтр реалізується за допомогою операції згортання з використанням ретельно підібраного набору значень для вектору $\vec{h}$. За правильного вибору елементів $\vec{h}$ можна спроектувати фільтри верхніх і нижніх частот, смугово-пропускні та смугово-загороджувальні фільтри. Як обговорювалося в попередніх абзацах, фільтри верхніх і нижніх частот пропускають високі та низькі частоти, відповідно, блокуючи інші частоти. **Смугопропускаючий фільтр** пропускає тільки частоти в межах зазначеного діапазону і блокує всі частоти поза цим діапазоном.

Смуговий загороджувальний фільтр пропускає всі частоти, окрім тих, які перебувають у зазначеному діапазоні.

Цілі високоефективного частотно-вибіркового фільтра – забезпечити мінімальне витіснення сигналу в смузі пропускання, ефективне блокування частот у смузі затримування і мінімальну ширину перехідної смуги.

Для створення високоефективного аналогового фільтра може знадобитися складна схема, що включає дорогі прецизійні компоненти. З іншого боку, високоефективний цифровий фільтр реалізується за допомогою всього лише операції згортання. Цифрова схема, що реалізує високоефективний фільтр нижніх частот з мінімальними спотвореннями в смузі пропускання і вузькою перехідною смугою, може вимагати довгий вектор $\vec{h}$, що містить, можливо, сотні або навіть тисячі елементів. Рішення про реалізацію такого фільтра в цифровому вигляді залежить від доступності недорогих ресурсів DSP, здатних виконувати операції MAC зі швидкістю, необхідною призначенням фільтра.

Швидке перетворення Фур'є

Перетворення Фур'є, назване на честь французького математика Жана-Батиста Жозефа Фур'є, розкладає сигнал у часовому просторі на набір синусоїдальних і косинусоїдальних хвиль різних частот і амплітуд. Початковий сигнал можна відновити шляхом підсумовування цих хвиль за допомогою процесу, що називається **оберненим перетворенням Фур'є**.

Процесори DSP працюють з відліками сигналів часової області, що відбираються через фіксовані інтервали. Завдяки такому відбору реалізація перетворення Фур'є в DSP називається **дискретним перетворенням Фур'є** (ДПФ). У загальному випадку ДПФ перетворює послідовність з *n* рівновіддалених один від одного тимчасових відліків функції в послідовність з *n* відліків ДПФ, рівновіддалених один від одного за частотою.

Кожен відлік ДПФ є комплексним числом, що складається з дійсного числа та уявного числа. **Уявне число** під час піднесення до квадрата дає від'ємний результат.

Ми не будемо тут заглиблюватися в математику уявних чисел. Альтернативний спосіб розгляду комплексного числа, що представляє частотну складову ДПФ (так звану **частотним елементом**), полягає в тому, щоб розглядати дійсну частину комплексного числа як множник для косинусоїдальної хвилі на частоті частотного елемента, а уявну частину – як множник для синусоїдальної хвилі на тій самій частоті. Підсумовування цих хвильових складових дає уявлення в часовій області для даного частотного елемента ДПФ.

Найпростіша реалізація алгоритму ДПФ для послідовності довжини *n* – це подвійний вкладений цикл, у якому кожен із циклів виконує ітерацію *n* раз. Якщо потрібно

збільшити довжину ДПФ, то кількість математичних операцій зростає пропорційно квадрату *n*. Наприклад, для обчислення ДПФ сигналу довжиною в 1000 відліків потрібно не менше мільйона операцій.

У 1965 р. Джеймс Кулі (James Cooley) з IBM і Джон Т'юкі (John Tukey) з Прінстонського університету опублікували роботу, яка описує комп'ютерну реалізацію більш ефективного алгоритму ДПФ, що став відомий як **швидке перетворення Фур'є** (ШПФ). Описаний ними алгоритм був спочатку винайдений німецьким математиком Карлом Фрідріхом Гауссом близько 1805 р.

Алгоритм ШПФ розбиває ДПФ на менші ДПФ, де довжини менших ДПФ можуть бути перемножені для отримання кількості відліків у вихідному ДПФ. Підвищення ефективності, що забезпечується алгоритмом ШПФ, є найбільшим, коли довжина послідовності для ДПФ дорівнює ступеню числа 2, що дає змогу виконувати рекурсивне розкладання надвоє по всій довжині ДПФ. ШПФ для 1024 точок вимагає всього лише кількох тисяч операцій порівняно з більш ніж мільйоном операцій для реалізації подвійного вкладеного циклу ДПФ.

Важливо розуміти, що ШПФ працює з тією самою послідовністю вхідних даних, що й ДПФ, і видає ті самі вихідні дані, що й ДПФ, але ШПФ робить це набагато швидше для довших послідовностей.

ШПФ використовується для розв'язання багатьох практичних завдань під час обробки сигналів. Ось деякі приклади.

- **Спектральний аналіз**. Вихідні дані ДПФ для послідовності відліків часової області – це набір комплексних чисел, що відображають амплітуду синусоїдальних і косинусоїдальних хвиль у діапазоні частот, який представляє сигнал. Ці амплітуди прямо вказують, які частотні компоненти присутні в сигналі зі значними рівнями, а які частоти мають менший або нехтувано малий внесок.

  Спектральний аналіз використовується в таких галузях, як обробка аудіосигналів, зображень і радіолокаційних сигналів. Лабораторні прилади, звані **аналізаторами спектра**, зазвичай використовуються для тестування і контролю роботи радіочастотних систем, таких як радіоприймачі та радіопередавачі.

  Аналізатор спектра показує періодично оновлюване зображення, що представляє частотний склад вхідного сигналу, отримане за допомогою ШПФ, обчисленого за відліками цього сигналу.

- **Банки фільтрів** являють собою набори окремих частотно-вибіркових фільтрів, кожен з яких обробляє окрему смугу частот. Повний набір фільтрів у банку охоплює весь діапазон частот вхідного сигналу. Прикладом банку фільтрів є **графічний еквалайзер**, який використовують у високоякісній аудіоапаратурі.

Банк фільтрів на основі БПФ корисний для розкладання декількох розділених за частотою каналів даних, переданих у вигляді єдиного комбінованого сигналу. У приймачі БПФ розділяє прийнятий сигнал на кілька смуг, кожна з яких містить незалежний канал передачі даних. Сигнал, що міститься у кожній із цих смуг, додатково обробляється для вилучення його вмісту.

Використання банків фільтрів на основі БПФ широко поширене в радіоприймачах для систем широкосмугової цифрової передачі даних, таких як цифрове телебачення та мобільний зв'язок 5G.

- **Стиснення даних**. сигнал можна стиснути до меншого розміру шляхом виконання ШПФ і відкидання частотних складових, які вважають несуттєвими. Решта частотних компонентів утворюють менший набір даних, який може бути додатково стиснутий з використанням стандартних методів кодування.

  Такий підхід називається **стисненням з втратами**, оскільки частина інформації вхідного сигналу втрачається. Стиснення з втратами зазвичай призводить до більшого ступеня стиснення сигналу порівняно зі **стисненням без втрат**. Алгоритми стиснення без втрат використовуються в ситуаціях, коли будь-яка втрата даних є неприпустимою, наприклад, при стисненні комп'ютерних файлів даних.

- **Дискретне косинусне перетворення** (ДКП) аналогічне за концепцією ДПФ, за винятком того, що замість розкладання вхідного сигналу на набір синусоїдальних і косинусоїдальних функцій, як у ДПФ, під час ДКП вхідний сигнал розкладають лише на косинусні функції, кожна з яких множиться на дійсне число. Обчислення ДКП можна прискорити за допомогою того ж підходу, що використовується в ШПФ для прискорення обчислення ДПФ. ДКП має ту цінну властивість, що в багатьох завданнях стиснення даних більша частина інформації про сигнал представлена в меншій кількості коефіцієнтів ДКП порівняно з альтернативними алгоритмами, такими як ДПФ. Це дає змогу відкинути більшу кількість менш значущих частотних складових, підвищивши тим самим ефективність стиснення даних.

Стиснення даних за допомогою ДКП використовують у багатьох прикладних галузях, з якими щодня взаємодіють користувачі комп'ютерів та аудіовізуальних розважальних систем, включно з аудіо у форматі MP3, зображеннями у форматі **JPEG** (Joint Photographic ExpepeПs Group) та відео у форматі **MPEG** (Moving PicShre Experts Group).

Процесори DSP часто використовують для розв'язання завдань, пов'язаних з обробленням одно- і двовимірних даних. Прикладами одновимірних даних є аудіосигнали та радіочастотні сигнали, які приймає приймач мобільного телефону. Одновимірні дані сигналів складаються зі значення одного відліку в кожен момент часу для кожного з декількох вхідних каналів.

Фотографічне зображення слугує прикладом двовимірних даних. Двовимірне зображення описується його шириною і висотою в пікселях і кількістю бітів, що представляють кожен піксель. Кожен піксель зображення відокремлений від навколишніх пікселів просторовим зсувом по горизонталі та вертикалі. Кожен піксель на зображенні (теоретично) відбирається в один і той самий момент часу.

Відеозображення являє собою тривимірну інформацію. Один зі способів визначення відео сегмента – це послідовність двовимірних зображень, що слідують одне за одним через рівні проміжки часу. Традиційні процесори DSP оптимізовані для роботи з двовимірними даними одного зображення, однак це не означає, що вони оптимальні і для обробки послідовних зображень з високою частотою оновлення. У наступному розділі описуються графічні процесори, тобто процесори, спеціалізовані для задоволення вимог, пов'язаних із синтезом і відображенням відео.

Обробка даних у графічних процесорах

Графічний процесор – це цифровий процесор, оптимізований для виконання математичних операцій, пов'язаних із генеруванням і візуалізацією графічних зображень для їх відображення на екрані комп'ютера. Основними областями застосування графічних процесорів є відтворення відео-записів і створення синтезованих зображень тривимірних сцен.

Продуктивність графічного процесора вимірюється з урахуванням роздільної здатності екрана (ширина і висота зображення в пікселях) і частоти оновлення зображення в кадрах на секунду. Відтворення відео та візуалізація сцен – це складні процеси жорсткого реального часу, в яких будь-яке порушення плавного і регулярного оновлення зображення через рівні проміжки часу, найімовірніше, буде сприйнято користувачами як неприйнятне підвисання графіки.

Як і відеокамери, які ми згадували раніше в цьому розділі, графічні процесори зазвичай представляють кожний піксель за допомогою трьох 8-бітних значень кольору, що вказують інтенсивність червоної, зеленої та синьої складових. Будь-який сприйманий колір можна отримати шляхом комбінування відповідних значень кожного з цих трьох кольорів. У кожному колірному каналі значення 0 вказує на відсутність кольору, а 255 – на максимальну інтенсивність. Чорний представлений трьома нульовими значеннями кольору (червоний, зелений, синій) = (0, 0, 0), а білий – (255, 255, 255). 24 біти колірних даних дають можливість відобразити понад 16 млн унікальних кольорів. Ступінь відмінності між сусідніми 24-бітними значеннями кольору зазвичай менший, ніж може розрізнити людське око.

У сучасних персональних комп'ютерах функції графічного процесора доступні в різних конфігураціях:

- графічна карта може бути встановлена в роз'єм PCIe;

- система може містити графічний процесор у вигляді однієї або кількох окремих інтегральних схем на основній платі процесора;
- функції графічного процесора можуть бути вбудовані в інтегральну схему центрального процесора.

Найпотужніші графічні процесори споживчого класу реалізовані у вигляді карт розширення з інтерфейсом PCIe. Ці високопродуктивні графічні процесори містять виділену графічну пам'ять і мають швидкий канал зв'язку з основним системним процесором (зазвичай з використанням роз'єму PCIe x16) для приймання команд і даних, що представляють відображувану сцену. Деякі графічні процесори підтримують використання декількох ідентичних карт в одній системі для створення сцен на одному графічному дисплеї. Ця технологія вимагає застосування окремої високошвидкісної шини, що з'єднує графічні процесори один з одним. Використання декількох графічних процесорів у системі забезпечує ефективне розпаралелювання графічної обробки.

Графічні процесори використовують концепцію паралелізму даних для одночасного виконання ідентичних обчислень над вектором елементів даних, створюючи відповідний вектор вихідних даних. Сучасні графічні процесори підтримують тисячі одночасних потоків, що виконуються одночасно, реалізуючи можливість візуалізації складних тривимірних зображень, що містять мільйони пікселів, зі швидкістю 60 кадрів на секунду або більше.

Архітектура типового графічного процесора складається з одного або декількох багатоядерних процесорів, кожен з яких підтримує багатопотокове виконання алгоритмів паралельної обробки даних. Інтерфейс між графічними процесорами та графічною пам'яттю оптимізовано таким чином, щоб забезпечити максимальну середню пропускну спроможність передавання даних, а не звести до мінімуму затримку доступу (що є метою проектування основної системної пам'яті). У графічних процесорах допускається деяке збільшення затримки заради досягнення максимальної швидкості потокової передачі даних між графічним процесором і його виділеною пам'яттю, оскільки максимальна пропускна здатність забезпечує досягнення максимально можливої частоти оновлення кадрів.

У комп'ютерних системах з менш жорсткими вимогами до продуктивності графічного оброблення, таких як комп'ютери бізнес-класу, дешевшим і часом цілком прийнятним рішенням є інтеграція менш продуктивного графічного процесора в одну мікросхему з основним процесором. Інтегровані графічні процесори можуть відтворювати потокове відео і реалізують певною мірою обмежений рівень можливостей тривимірної візуалізації сцен порівняно з графічними процесорами більш високого класу.

Замість виділеної графічної пам'яті інтегровані графічні процесори використовують для візуалізації графіки частину системної пам'яті. Такий підхід призводить до

зниження продуктивності, однак подібні системи забезпечують достатню продуктивність оброблення графіки для розв'язання більшості офісних і домашніх завдань.

Інтелектуальні пристрої, такі як мобільні телефони та планшети, також містять графічні процесори, що забезпечують ті самі можливості відтворення відео та тривимірної візуалізації сцен, що й більші персональні комп'ютерні системи. Вимоги компактності та зниженого енергоспоживання неминуче обмежують продуктивність графічних процесорів мобільних пристроїв. Проте сучасні смартфони та планшети повною мірою здатні відтворювати потокове відео високої чіткості і відображати складну ігрову графіку.

Графічні процесори як процесори обробки даних

Протягом багатьох років архітектури графічних процесорів розроблялися виключно для підтримки обчислювальних потреб візуалізації тривимірних сцен у реальному часі. Останніми роками користувачі та виробники графічних процесорів дедалі частіше визнають, що ці пристрої насправді є мініатюрними суперкомп'ютерами, придатними для розв'язання набагато ширшого спектра завдань. Сучасні графічні процесори забезпечують швидкість виконання операцій з плаваючою комою, що вимірюється трильйонами операцій з плаваючою комою на секунду **(терафлопс)**. Станом на 2021 р. типовий високопродуктивний графічний процесор підтримував продуктивність операцій із плаваючою комою на рівні десятків терафлопс, і міг виконувати математичні алгоритми паралельного оброблення даних у сотні разів швидше, ніж стандартний настільний комп'ютер.

Використовуючи переваги величезної потужності паралельних обчислень, доступної у високопродуктивних графічних процесорах, виробники цих пристроїв надають програмні інтерфейси і розширені апаратні можливості для реалізації більш узагальнених алгоритмів. Безумовно, графічні процесори, навіть із поліпшеннями для підтримки вимог обчислень загального призначення, по-справжньому ефективні тільки під час прискорення алгоритмів, які припускають розпаралелювання обробки даних.

Далі наведено деякі сфери застосування, які виявилися придатними для прискорення операцій за допомогою графічних процесорів.

Великі дані

У таких різнопланових сферах, як моделювання клімату, картування генів, бізнес-аналітика та аналіз сейсмічних даних, сфери розв'язуваних проблем мають загальну потребу в максимально ефективному аналізі величезних обсягів даних, часто вимірюваних у **терабайтах** (**Тбайт**) або **петабайтах** (**Пбайт**) (1 Пбайт = 1024 Тбайт).

У багатьох випадках алгоритми такого аналізу перебирають великі набори даних у пошуках тенденцій, кореляцій і складніших зв’язків між елементами даних, які на перший погляд можуть здатися несумісними і не пов'язаними між собою масивами вибірок.

Донедавна аналіз таких наборів даних за відповідного рівня деталізації часто відкидався як нездійсненний через надто великий час, який не потрібен для виконання такого опрацювання. Однак сьогодні багато додатків для опрацювання великих даних дають результати за цілком розумні проміжки часу, комбінуючи використання графічних процесорів (часто на машинах, що містять кілька взаємопов'язаних графічних процесорів) і розподіляючи завдання між кількома комп'ютерними системами в хмарному середовищі. У наші дні використання декількох комп'ютерів, кожен з яких містить кілька графічних процесорів, для виконання алгоритмів з високим ступенем паралелізму над величезним набором даних може вимагати напрочуд низьких витрат у порівнянні з витратами, які зазвичай пов'язують із суперкомп'ютерами.

Глибоке навчання

Глибоке навчання – це категорія методів штучного інтелекту (ШІ), яка використовує багаторівневі мережі штучних нейронів для моделювання фундаментальних операцій клітин людського мозку. **Біологічний нейрон** – це тип нервової клітини, яка обробляє інформацію. Нейрони пов'язані між собою за допомогою **синапсів** і використовують для обміну інформацією електрохімічні імпульси. Під час навчання людський мозок налаштовує зв'язки між нейронами, щоб кодувати засвоювану інформацію для подальшого вилучення. Людський мозок містить десятки мільярдів нейронів.

Штучні нейронні мережі (artificial neural network, ANN) використовують програмну модель поведінки нейронів для імітації процесів навчання та вилучення інформації, що протікають у людському мозку. Кожен штучний нейрон отримує вхідні дані від (можливо) багатьох інших нейронів і обчислює один числовий результат. Деякі нейрони керуються безпосередньо вхідними даними, що підлягають обробці, а інші виробляють вихідні дані, одержувані в результаті обчислень ANN. Кожен канал зв'язку між нейронами має пов'язаний із ним ваговий коефіцієнт, який являє собою звичайне число, що помножує силу сигналу, що проходить цим шляхом. Виражені у вигляді числа вхідні дані для нейрона – це сума одержуваних ним вхідних сигналів, кожен з яких помножений на вагу відповідного шляху.

Нейрон обчислює вхідний сигнал, використовуючи формулу, звану **функцією активації**. Вона визначає ступінь впливу вхідних даних на "спрацьовування" кожного нейрона.

На рис. 6.4 наведено приклад одного нейрона, який підсумовує вхідні дані від трьох інших нейронів ($N_1 – N_3$), множачи їх на вагові коефіцієнти ($W_1 – W_3$). Отриману суму передають у функцію активації F(x), що виробляє вихідний сигнал нейрона.

Використання трьох вхідних сигналів у цьому прикладі є довільним; у реальних задачах кожен нейрон може отримувати вхідні дані від будь-якої кількості інших нейронів.

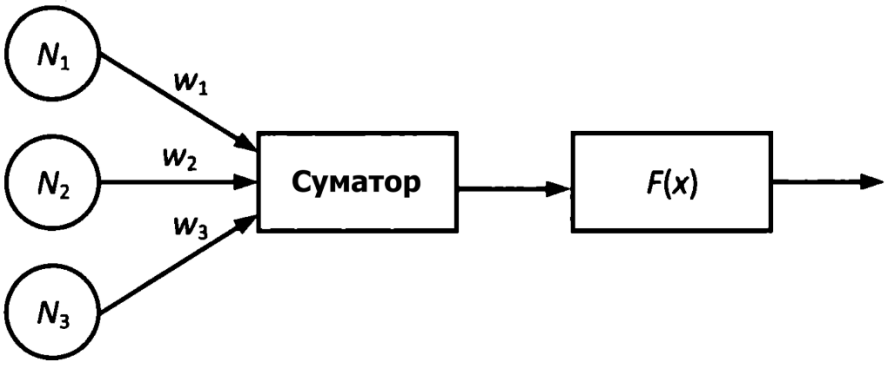


Рис 6.4. *Нейрон, який отримує вхідні сигнали від трьох інших нейронів*

ШНМ має багаторівневу організаіпію, в якій за першим рівнем, так званим **вхідним рівнем**, слідує один або кілька внутрішніх рівнів (так званих **прихованих рівнів**), після яких слідує **вихідний рівень**. Деякі ШНМ організовані у вигляді послідовності потоків даних, що протікають від входу до виходу. Мережу такої конфігурації називають **мережею прямого поширення**. Інші конфігурації передбачають зворотний зв'язок від деяких нейронів до нейронів на попередніх рівнях. Мережу такої конфігурації називають **рекурентною мережею**.

На рис. 6.5 показано приклад простої мережі прямого поширення з трьома вхідними (input) нейронами, прихованим (hidden) рівнем, що складається з чотирьох нейронів, і двома вихідними (output) нейронами. Ця мережа є повнозв'язною. Це означає, що кожен нейрон на вхідному і прихованому рівнях пов'язаний з усіма нейронами на наступному рівні. Вага з'єднань на цій діаграмі не відображенна.

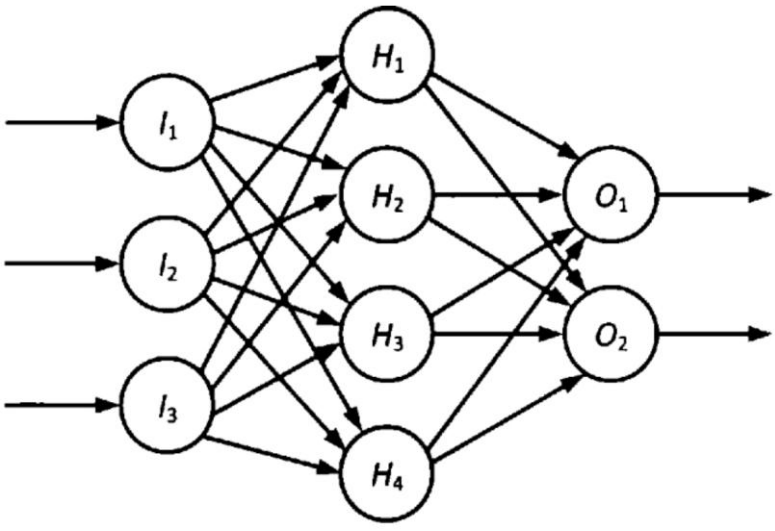


Рис 6.5. *Трирівнева мережа прямого поширення*

Навчання ШНМ полягає в налаштуванні зважених зв'язків між нейронами таким чином, щоб під час подання певного набору вхідних даних отримувався бажаний результат.

Застосовуючи відповідний алгоритм навчання, ШНМ можна навчити за допомогою набору даних, що складається з відомих завідомо правильних вихідних даних для широкого спектра вхідних даних. Навчання великої і складної ШНМ виконанню складної задачі, як-от керування автомобілем або гра в шахи, вимагає величезної кількості навчальних ітерацій, узятих з дуже великого набору даних. Під час навчання кожна ітерація вносить невеликі коригування у вагові коефіцієнти всередині мережі, поступово приводячи мережу в стан конвергенції. Після повної конвергенції мережа вважається навченою і може бути використана для отримання вихідних даних при поданні нових вхідних даних. Іншими словами, мережа узагальнює інформацію, яку вона засвоїла під час навчання, і застосовує ці знання до нових ситуацій.

Обробка на графічному процесорі дуже добре підходить для ШНМ завдяки їхньому паралельному характеру. Людський мозок фактично являє собою комп'ютер із мільярдами незалежних процесорних блоків і масовим паралелізмом. Цю форму паралелізму використовують на етапі навчання ШНМ для прискорення конвергенції мережі шляхом паралельного виконання обчислень, пов'язаних із безліччю штучних нейронів.

У наступному розділі буде представлено деякі приклади типів комп'ютерних систем, реалізованих на основі архітектурних концепцій, які було описано в цьому розділі.

Приклади специфічних архітектур

У цьому розділі розглядаються деякі конфігурації обчислювальних систем, спеціалізовані для вирішення певних завдань, і виділяються особливі вимоги до архітектури, властиві кожній з них. Ми розглянемо такі конфігурації.

- **Сервер для хмарних обчислень.** Кілька постачальників пропонують клієнтам доступ до обчислювальних платформ через Інтернет. Користувачі можуть завантажувати програмні додатки на відповідний хмарний сервер і виконувати будь-які види обчислень за своїм бажанням. Як правило, ці сервіси виставляють рахунки своїм клієнтам на підставі типу та кількості виділених обчислювальних ресурсів і тривалості їх використання. Перевагою для клієнта є те, що за ці послуги не треба платити, коли вони не використовуються.

  На вищому рівні продуктивності клієнти можуть використовувати сервери, що містять кілька з'єднаних одна з одною карт із графічними процесорами, для виконання великомасштабних обчислень з інтенсивним використанням операцій із плаваючою комою на величезних масивах даних. У контексті хмарних обчислень цілком природним і часто економічно вигідним є підхід із

розбивкою обчислювальних завдань на більш дрібні частини, які можуть виконуватися паралельно на кількох серверах із графічними процесорами.

Завдяки цьому організації і навіть приватні особи з обмеженими фінансовими можливостями можуть використовувати обчислювальні ресурси, які ще кілька років тому були виключною прерогативою уряду, великого бізнесу і дослідницьких університетів, що мають у своєму розпорядженні засоби для створення суперкомп'ютерів.

- **Настільний комп'ютер бізнес-класу**. Керівництво відділів інформаційних технологій комерційних підприємств прагне забезпечити співробітників обчислювальними ресурсами, необхідними для виконання їхньої роботи, з найменшими витратами. Більшості офісних працівників не потрібна виняткова графічна або обчислювальна продуктивність, однак їхні комп'ютерні системи повинні відповідати помірним вимогам щодо відтворення відео презентацій для таких цілей, як навчання.

  Для бізнес-користувачів зазвичай більш ніж достатньо графічного процесора, інтегрованого з сучасним центральним процесором. За розумну ціну бізнес-покупці можуть придбати комп'ютерні системи з процесорами середнього рівня продуктивності з інтегрованим графічним процесором. Подібні системи забезпечують повну підтримку сучасних операційних систем і стандартних офісних додатків, таких як текстовий редактор, електронна пошта та електронні таблиці. Якщо виникне потреба в розширенні можливостей системи з оброблення графіки, найпростіший спосіб модернізації – встановлення карти графічного процесора в роз'єм розширення.

- **Високопродуктивний ігровий комп'ютер**. Любителям новітніх комп'ютерних 3D-ігор потрібен графічний процесор із найвищим рівнем продуктивності для відображення деталізованих сцен із високою роздільною здатністю за максимально можливої частоти кадрів. Ці користувачі готові вкласти свої кошти в потужний, дорогий, такий, що потребує потужного живлення графічний процесор (або навіть кілька процесорів) для досягнення максимально можливої графічної продуктивності.

  Не менш важливою особливістю такого ігрового комп'ютера, крім потужного графічного процесора, є наявність швидкого системного процесора. Системний і графічний процесори працюють спільно, використовуючи високошвидкісний інтерфейс (зазвичай PCIe x16), що з'єднує їх, для обчислення положення і напряму огляду спостерігача сцени, а також типу, місця розташування, візуальних характеристик і орієнтації всіх об'єктів у сцені. Системний процесор передає цю геометричну інформацію графічному процесору, який виконує математичні операції, необхідні для створення реалістичного зображення на

екрані. Цей процес повторюється з частотою, достатньою для плавного відображення складних, швидко мінливих сцен.

- **Смартфон високого класу**. Сучасні смартфони поєднують високо-продуктивні обчислювальні можливості та високоякісні графічні дисплеї з жорсткими обмеженнями щодо енергоспоживання і тепловиділення. Користувачі вимагають швидкої, плавної та яскравої графіки для ігор і перегляду відео, але при тому вони не згодні миритися з малим часом автономної роботи або сильним нагріванням пристрою.

  Дисплеї сучасних телефонів містять мільйони повнокольорових пікселів, до 12 Гбайт оперативної пам'яті та до 1 Тбайт флеш-пам'яті. Ці пристрої зазвичай оснащуються двома камерами з високою роздільною здатністю (передньою і задньою), здатними робити знімки і записувати відео. Телефони високого класу містять 64-розрядний багатоядерний процесор з інтегрованим графічним процесором, а також безліч функцій, що забезпечують оптимальне поєднання енергоспоживання та високої продуктивності.

  Архітектури смартфонів містять процесори DSP для розв'язання таких завдань, як кодування і декодування голосових аудіосигналів під час телефонних розмов та обробка прийнятих і переданих радіочастотних сигналів, які використовуються різними приймачами-передавачами телефону. Типовий телефон підтримує цифровий стільниковий зв'язок, Wi-Fi, Bluetooth і **ближній радіозв'язок** (near-field comшunication, NFC). Сучасні смартфони – це потужні обчислювальні платформи, оснащені широким спектром засобів зв'язку та оптимізовані для роботи від акумулятора.

У цьому розділі ми розглянули архітектури комп'ютерних систем, що представляють лише малу частину сучасних і перспективних областей застосування обчислювальної техніки. Незалежно від призначення комп'ютерної системи, чи то настільний офісний комп'ютер, чи то смартфон, чи то система керування пасажирським авіалайнером, у процесі її проектування та реалізації застосовується єдиний набір загальних архітектурних принципів.

Резюме

У цьому розділі було розглянуто кілька спеціалізованих галузей обчислень, включно із системами реального часу, цифровою обробкою сигналів і обробкою графічної інформації Після ознайомлення з цим розділом у вас має скластися більш повне уявлення про особливості сучасних комп'ютерів, включно з роботою в реальному часі, обробленням аналогових сигналів і графіки в таких галузях, як ігри, голосовий зв'язок, відтворення відео та використання графічних процесорів як мініатюрних суперкомп'ютерів. Ці можливості є важливими розширеннями основних

обчислювальних завдань, які виконує центральний процесор, чи то хмарний сервер, чи то настільний комп'ютер, чи то смартфон.

У наступному розділі ми обговоримо сучасні процесорні архітектури, зокрема фон-нейманівську, гарвардську та модифіковану гарвардську. У ньому буде розглянуто використання віртуальної пам'яті зі сторінковою організацією, а також можливості та функції модуля керування пам'яттю.

Вправи

1. **Частотно-монотонне планування (RMS)** – це алгоритм призначення пріоритетів потокам у додатках жорсткого реального часу з витісненням, у яких потоки виконуються періодично. RMS призначає найвищий пріоритет потоку з найкоротшим періодом виконання, наступний за значущістю пріоритет – потоку з наступним найбільш коротшим періодом виконання тощо. Система RMS є диспетчерською, тобто всі її завдання гарантовано вкладаються у встановлені строки (за умови, що між потокові взаємодії або інші дії, такі як переривання, не спричиняють затримок опрацювання), якщо виконується наступна умова:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right).$$

Ця формула виражає максимальну частку доступного часу обробки, що може бути витрачена л потоками. У цій формулі C_i – це максимальний час виконання, необхідний для потоку i, а T_i, – період виконання потоку i.

Чи є наступна система, що складається з трьох потоків, підконтрольною?

Потік	Час виконання C_i, мс	Період виконання T_i, мс
Потік 1	50	100
Потік 2	100	500
Потік 3	120	1000

2. Широко використовувана форма одновимірного дискретного косинусного перетворення (ДКП) виражається такою формулою:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}(n + \frac{1}{2}k\right].$$

У цій формулі k, індекс коефіцієнта ДКП, змінюється від 0 до N – 1. Напишіть програму для обчислення ДКП такої послідовності:

$$x = \{0.5, 0.2, 0.7, -0.6, 0.4, -0.2, 1.0, -0.3\}.$$

Косинусні члени у формулі залежать тільки від індексів *n* і *k* і не залежать від послідовності вхідних даних *x*. Це означає, що косинусні члени можна обчислити один раз і зберегти у вигляді констант для подальшого використання. Якщо зробити це як підготовчий етап, то обчислення кожного коефіцієнта ДКП зводиться до послідовності операцій MAC.

Ця формула являє собою неоптимізовану форму обчислення ДКП, що вимагає N^2 ітерацій операції MAC для обчислення всіх коефіцієнтів ДКП у кількості *N*.

3. В якості функції активації в ІНС часто використовується гіперболічний тангенс. Його функція визначається наступним чином:

$$th(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Дано нейрон із входами від трьох попередніх нейронів, показаний на рис. 6.4. Обчисліть вихідний сигнал нейрона з гіперболічним тангенсом як функцією активації *F(x)*, використовуючи такі вихідні сигнали попередніх нейронів і ваги відповідних шляхів:

Нейрон	Вихідний сигнал нейрона	Вага
N_1	0,6	0,4
N_2	– 0,3	0,8
N_3	0,5	– 0,2

7

Архітектура процесора і пам'яті

У цьому розділі більш докладно розглядаються архітектури сучасних процесорів, зокрема фон-нейманівська, гарвардська та модифікована гарвардська, а також галузі обчислень, у яких, як правило, застосовується кожна з цих архітектур. Представлено концепції та переваги віртуальної пам'яті зі сторінковою організацією, широко використовуваної в обчислювальних платформах споживчого та комерційного призначення, а також у мобільних пристроях. Ми розглянемо практичні деталі керування пам'яттю на прикладі реальних систем – Windows NT і пізніших версій Windows. Розділ завершується обговоренням особливостей і функцій блоку керування пам'яттю.

Після ознайомлення з даним розділом ви отримаєте уявлення про ключові особливості архітектур сучасних процесорів і використання фізичної та віртуальної пам'яті. Ви також ознайомитеся з перевагами сторінкової організації пам'яті та функціями блока керування пам'яттю.

У цьому розділі розглядаються такі теми:

- фон-нейманівська, гарвардська та модифікована гарвардська архітектури;
- фізична та віртуальна пам'ять;
- віртуальна пам'ять зі сторінковою організацією;
- блок керування пам'яттю.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою https://github.com/PacktPub1ishing/Modern-Computer-Architecture-and-OrganizationSecond-Edition.

Фон-нейманівська, гарвардська та модифікована гарвардська архітектури

У попередніх розділах ми коротко торкнулися історії та сучасних областей застосування процесорних архітектур — фон-нейманівської, гарвардської та модифікованої гарвардської. У цьому розділі ми розглянемо ці конфігурації більш детально та вивчимо області обчислень, в яких зазвичай застосовується кожна із зазначених архітектур.

Фон-нейманівська архітектура

Архітектура, що дістала назву фон-нейманівської, була представлена Джоном фон Нейманом у 1945 р. Ця процесорна конфігурація складається з пристрою керування, арифметико-логічного пристрою (AJIY), набору регістрів і області пам'яті, яка містить програмні інструкції та дані. Ключова особливість, що відрізняє фон-нейманівську архітектуру від гарвардської, полягає у використанні єдиної області пам'яті для програмних інструкцій і даних, з якими працюють інструкції. Програмістам принципово просто, а розробникам схем відносно простіше розмістити весь код і всі дані, необхідні програмі, в одній області пам'яті.

На рис. 7.1 показано елементи фон-нейманівської архітектури.

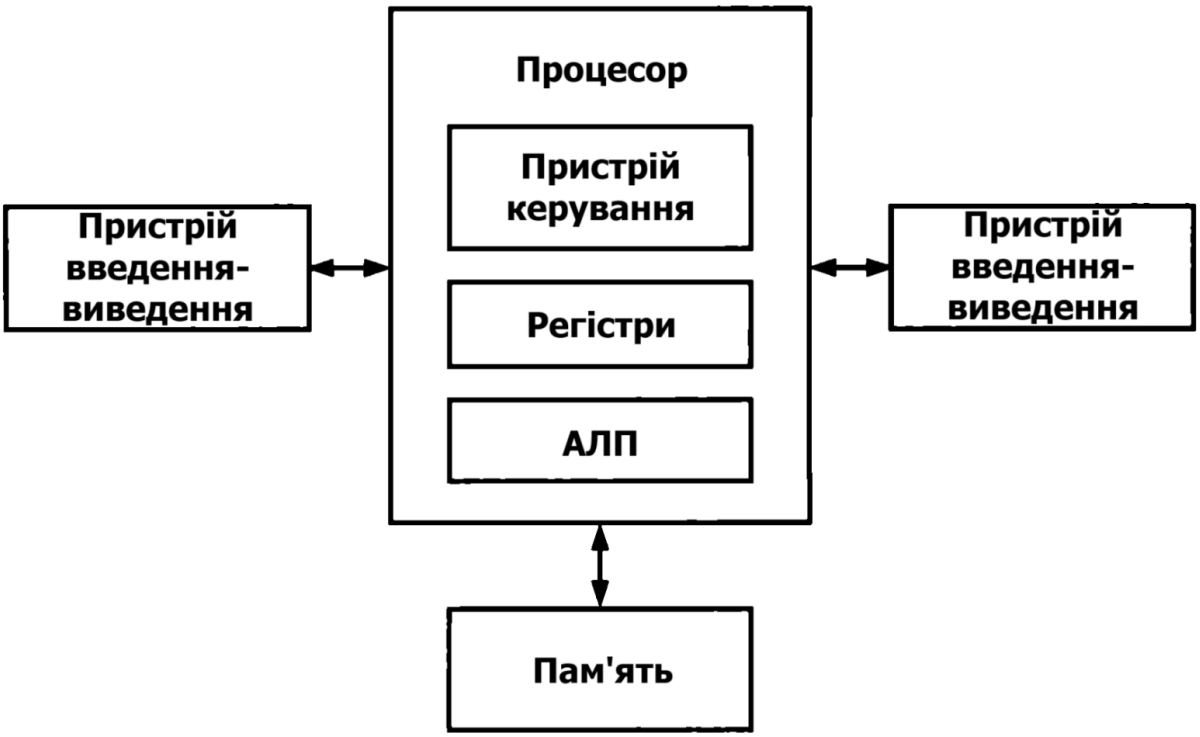


Рис 7.1. *Фон-нейманівська архітектура*

Архітектурний підхід з єдиною пам'яттю спростив проектування та виготовлення процесорів і комп'ютерів перших поколінь, однак використання спільної пам'яті для

програм і даних створило певні проблеми, пов'язані з продуктивністю систем і, останніми роками, з їхньою безпекою.

Ось деякі з найсуттєвіших проблем.

- **Вузьке місце архітектури фон Неймана.** Використання єдиного інтерфейсу між процесором і основною пам'яттю для доступу до інструкцій і даних часто вимагає декількох циклів пам'яті для витягання інструкції процесора і доступу до необхідних даних. Якщо безпосереднє значення зберігається поряд з опкодом інструкції, зниження продуктивності через вузьке місце може бути незначним або взагалі відсутнім, оскільки, принаймні в деяких випадках, безпосереднє значення завантажується разом з опкодом за один цикл доступу до пам'яті. Однак більшість програм витрачають значну частину свого часу на роботу з даними, що зберігаються у виділеній області пам'яті окремо від програмних інструкцій. У такій ситуації для витягнення опкоду і будь-яких необхідних елементів даних потрібно кілька операцій доступу до пам'яті.

  Використання кеш-пам'яті для програмних інструкцій і даних, що детально розглядається в *розділі 8*, може значно пом'якшити це обмеження. Однак під час роботи з послідовностями інструкцій та об'єктами даних, розмір яких перевищує обсяг кеш-пам'яті, перевага кешування зменшується і, ймовірно, навіть на значну величину. Неможливо уникнути того факту, що розміщення коду і даних в одній і тій самій ділянці пам'яті зі спільним каналом обміну даними з процесором іноді буде обмежувати продуктивність системи.

- **Аспекти безпеки фон-нейманівської архітектури.** Використання єдиної області пам'яті для коду і даних відкриває творчим програмістам можливість зберігати послідовності інструкцій у пам'яті під виглядом даних, а потім надсилати ці інструкції в процесор на виконання. Програми, які записують код у пам'ять, а потім виконують його, реалізують **самомодифікуючий код**. Крім складності пошуку та усунення помилок (оскільки багато засобів налагодження програмного забезпечення ґрунтуються на припущенні, що програма в пам'яті містить усі інструкції, які були спочатку зібрані в неї під час компіляції), цю можливість роками використовували хакери в зловмисних цілях.

- **Переповнення буфера** – гнітюче поширений недолік у широковживаних програмних рішеннях, таких як операційні системи, веб-сервери та бази даних. Переповнення буфера відбувається, коли програма запитує вхідні дані і зберігає їх у буфері даних фіксованої довжини. Якщо в коді не передбачена перевірка довжини даних, що вводяться, то користувач може ввести послідовність, довжина якої перевищує доступний простір для зберігання. Коли це відбувається, додаткові дані перезаписують вміст комірок пам'яті, призначених для інших цілей.

Якщо буфер, що перезаписується, зберігається в стеку програми, то винахідливий користувач може ввести довгу послідовність, яка може перезаписати адресу повернення виконуваної на даний момент функції, яка може зберігатися в тому ж стеку.

Ретельно продумавши вміст послідовності, що вводиться, зловмисник може перехопити керування виконуваним додатком і направити його на виконання будь-якої бажаної послідовності інструкцій. Для цього хакер повинен підготувати послідовність, яка переповнює буфер введення, перезаписує адресу повернення функції іншою, ретельно підібраною адресою та записує в пам'ять послідовність інструкцій, виконання якої запускається за цією адресою. Послідовність інструкцій, введена зловмисником, починає виконуватися, коли функція, яка спочатку запитувала користувацьке введення, здійснює повернення, передаючи керування коду хакера. Це дає змогу хакеру "заволодіти" комп'ютером.

Протягом багатьох років різні спроби розв'язати проблему переповнення буфера забирали величезну кількість часу в дослідників комп'ютерної безпеки, починаючи з першого випадку широкого розповсюдження атаки такого типу в 1988 р. Виробники процесорів і розробники операційних систем впровадили безліч функцій для боротьби з атаками переповнення буфера. Серед них рішення з такими назвами, як **запобігання виконанню даних** (data execution prevention, DEP) і **випадковий розподіл адресного простору** (address space layout randomization, ASLR). Ці виправлення продемонстрували деякий ступінь ефективності, однак фундаментальною особливістю процесора, яка створює можливість зловживань такого типу, є використання однієї й тієї самої ділянки пам'яті для програмних інструкцій і даних у фон-нейманівській архітектурі.

Гарвардська архітектура

Гарвардська архітектура була спочатку реалізована в комп'ютері Harvard Mark I у 1944 р. У суворій гарвардській архітектурі для програмних інструкцій і для даних відведено окремі адресні простори та шини доступу до пам'яті. Безпосередньою перевагою такої конфігурації є можливість одночасного доступу до інструкцій і до даних, що дає змогу реалізувати певну форму паралелізму. Звісно, це поліпшення досягається за рахунок суттєвого дублювання кількості адресних рядків, ліній передавання даних і керуючих сигналів, які мають бути реалізовані в процесорі для доступу до обох областей пам'яті.

На рис. 7.2 показано схему процесора, що реалізує гарвардську архітектуру.

Гарвардська архітектура може забезпечити вищий рівень продуктивності за рахунок розпаралелювання доступу до інструкцій і даних. Ця архітектура також усуває весь клас проблем безпеки, пов'язаних зі зловмисним виконанням програмних інструкцій,

прихованих під виглядом даних, за умови, що пам'ять інструкцій не може бути змінена програмними інструкціями.

Цей підхід передбачає, що пам'ять програм завантажена інструкціями належним чином.

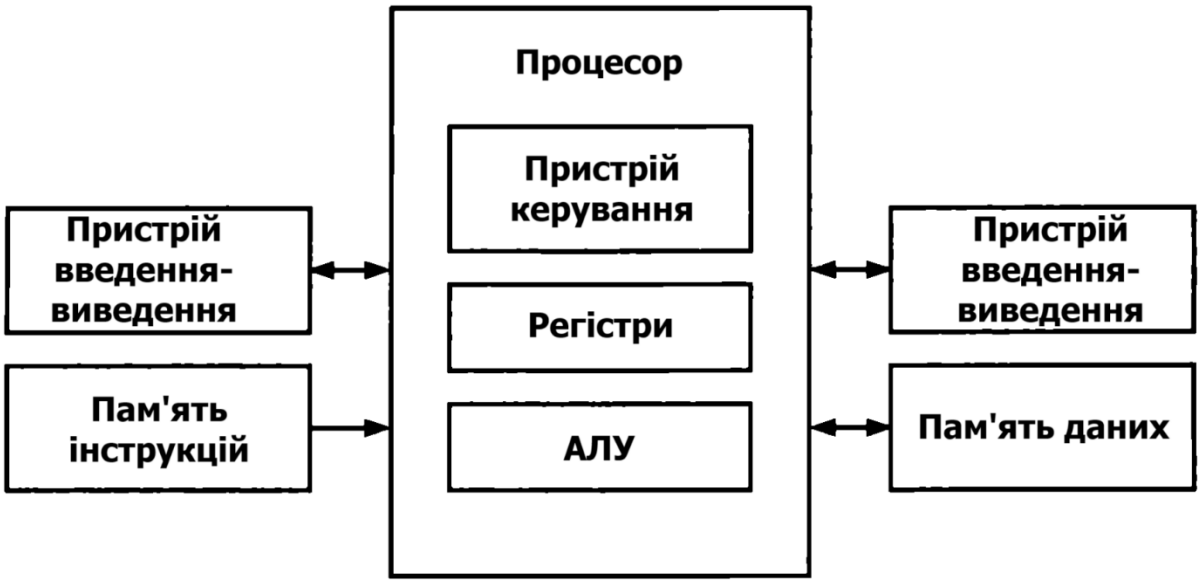


Рис 7.2. *Гарвардська архітектура*

Якщо озирнутися назад, маючи в своєму розпорядженні знання про поширення загроз безпеці, що ґрунтуються на особливостях архітектури фон Неймана, то є привід замислитися, чи не було б значно краще для всієї галузі інформаційних технологій, якби ще на ранніх етапах було досягнуто згоди щодо впровадження гарвардської архітектури з повним розподілом ділянок пам'яті коду та даних, незважаючи на пов'язані з цим витрати.

На практиці сувора гарвардська архітектура зрідка застосовується в сучасних комп'ютерах. Зазвичай використовується кілька варіантів гарвардської архітектури, які в сукупності називаються модифікованими гарвардськими архітектурами. Ці архітектури є темою наступного розділу.

Модифікована гарвардська архітектура

У комп'ютерах, розроблених на основі модифікованої гарвардської архітектури, реалізовано, як правило, деякий ступінь поділу між програмними інструкціями і даними. Це знижує вплив вузького місця архітектури фон Неймана і дає змогу усунути деякі проблеми безпеки, про які ми вже говорили. Однак поділ між інструкціями і даними зрідка буває абсолютним. У системах з модифікованою гарвардською архітектурою використовуються окремі області пам'яті програмних інструкцій і пам'яті даних, але ці процесори зазвичай підтримують деякі засоби для збереження даних у пам'яті програм і, в окремих випадках, для збереження інструкцій у пам'яті даних.

На рис. 7.3 показано модифіковану гарвардську архітектуру, що представляє багато реальних комп'ютерних систем.

Як ми бачили в попередньому розділі, процесори цифрової обробки сигналів (digital signal processors, DSP) отримують значні переваги від використання архітектури, подібної до гарвардської.

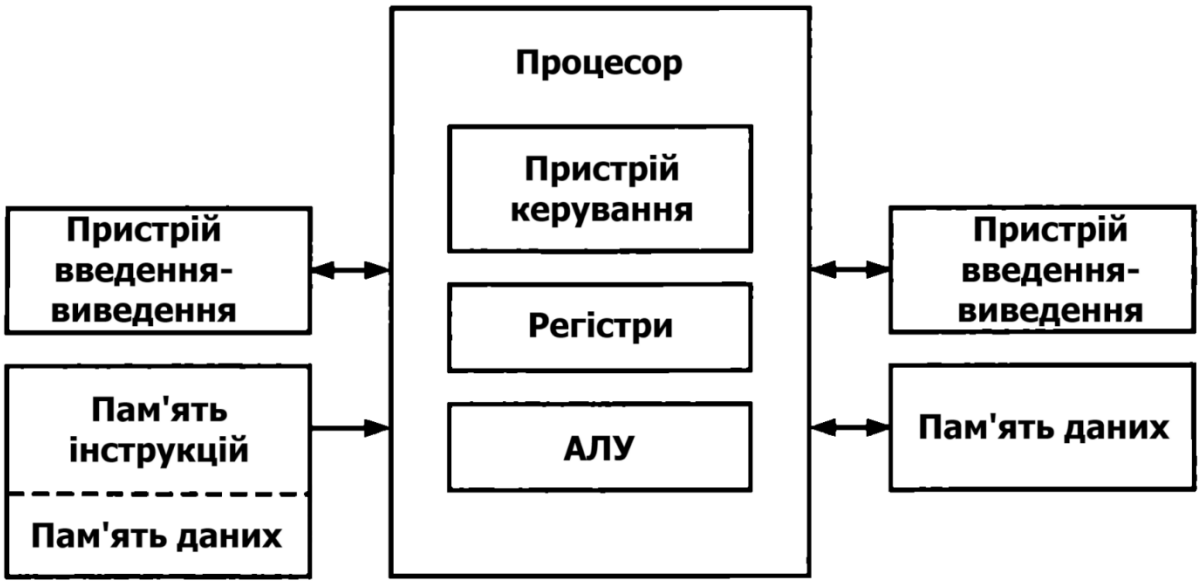


Рис 7.3. *Модифікована гарвардська архітектура*

Зберігаючи один числовий вектор у пам'яті інструкцій і другий вектор – у пам'яті даних, DSP може виконати одну операцію множення з накопиченням (multiply accumulate, MAC) за один тактовий цикл процесора. У цих системах область пам'яті інструкцій і елементи даних, що містяться в ній, зазвичай доступні тільки для читання. На це вказує односпрямована стрілка, що з'єднує пам'ять інструкцій із процесором на рис. 7.3. Тому для збереження в області пам'яті інструкцій підходять тільки значення постійних даних.

Крім DSP, більшість сучасних процесорів загального призначення містять окремі області кеш-пам'яті для інструкцій і даних, реалізуючи тим самим важливі особливості гарвардської архітектури. Такі процесорні архітектури, як 386 і ARM, підтримують паралельний і незалежний доступ до інструкцій і даних, коли запитувані елементи знаходяться на першому рівні кеш-пам'яті. Якщо пошук у вбудованій кеш-пам’яті не увінчався успіхом, процесор повинен звернутися до основної пам'яті загальною фон-нейманівською шиною, що займає значно більше часу.

На практиці деталі реалізації конкретного процесора з погляду особливостей фон-нейманівської та гарвардської архітектур зрідка мають значення для працівників програмного забезпечення, за винятком міркувань продуктивності. Програмісти зазвичай розробляють програми на обраній ними мові високого рівня, а деталі операцій, пов'язаних з розподілом даних та інструкцій по відповідних ділянках пам'яті, реалізує компілятор чи інтерпретатор.

У наступному розділі обговорюються переваги віртуалізації пам'яті.

Фізична та віртуальна пам'ять

Пристрої пам'яті в комп'ютерах можна поділити на пристрої **оперативної пам'яті** (оперативний запам'ятовувальний пристрій, ОЗП – random access memory, RAM), читання з яких і запис у які виконуються довільно, і пристрої **постійної пам'яті** (постійного запам'ятовуючого пристрою, ПЗП – read-only memory, ROM), дані з яких, як зрозуміло з їхньої назви, можна тільки зчитувати, але не записувати. Деякі типи пристроїв пам'яті, такі як флеш-пам'ять і **електрично стирається програмований постійний запам'ятовуючий пристрій** (ЕСППЗП), представляють золоту середину, коли вміст пристроїв може бути змінено, але не так легко або не так швидко або не так багато разів, як у випадку звичайної оперативної пам'яті.

Пристрої пам'яті в комп'ютері мають бути налаштовані таким чином, щоб гарантувати, що кожний пристрій займає унікальну ділянку системного адресного простору, щоб процесор міг одержувати доступ до кожного з (можливих) кількох ОЗП і ПЗП шляхом вибору відповідних адресних рядків. Сучасні комп'ютерні системи зазвичай виконують такий розподіл адресного простору автоматично, ґрунтуючись на роз'ємі, у який встановлено пристрій пам'яті.

Програмне забезпечення, що працює на ранніх комп'ютерних системах, а також на відносно нескладних сучасних комп'ютерах і вбудовуваних процесорах (таких як системи на базі 6502), для виконання операцій читання та запису використовує в програмних інструкціях адреси в ОЗП і ПЗП.

Наприклад, така інструкція процесора 6502, як JMP $1000, наказує процесору завантажити в покажчик інструкцій шістнадцятирічне значення $1000 і виконати інструкцію в цьому осередку пам'яті. Під час опрацювання цієї інструкції пристрій керування процесора 6502 поміщає значення $1000 на 16 адресних рядків процесора і зчитує байт за відповідною адресою пам'яті. Цей байт інтерпретується як опкод наступної інструкції, яка має бути виконана. Аналогічним чином, завантаження байта з пам'яті за допомогою такої інструкції, як LDA $0200, поміщає значення $0200 на адресні рядки та копіює байт, що зберігається за цією адресою, у регістр A.

У системах, що використовують фізичну адресацію, адреси пам'яті в інструкціях є фактичними адресами зазначеної інструкції або елемента даних. Це означає, що адреса пам'яті, яка міститься в інструкції, є тією ж самою адресою, яка використовується для електричного доступу до відповідної комірки в пристрої пам'яті.

Цей архітектурний підхід концептуально простий для реалізації в процесорах, але в сценарії, що містить кілька програм, які виконуються одночасно (який називають **мультипрограмним режимом**), складність розробки програмного забезпечення може швидко стати надмірною. Якщо кожна з кількох програм розробляється

ізольовано від інших (наприклад, у випадку з кількома незалежними розробниками), має бути якийсь спосіб розподілу доступних адресних просторів ОЗП і ПЗП для окремих програм, щоб гарантувати одночасне виконання кількох програм (можливо, у контексті ОСРЧ) без взаємних конфліктів під час доступу до пам'яті.

Однією з добре відомих ранніх спроб підтримки виконання кількох програм в одному адресному просторі на ПК є реалізована в операційній системі MS-DOS концепція **резидентної програми** (terminate and stay resident, TSR). Резидентні програми резервують ділянку пам'яті і завантажують у неї свій код, а потім повертають керування операційній системі. Користувачі можуть продовжувати працювати з системою у звичайному режимі, завантажуючи і використовуючи інші додатки (по одному, звісно), але вони також можуть отримати доступ до резидентної програми в міру необхідності, зазвичай за допомогою спеціальної комбінації клавіш. Можна одночасно завантажити в пам'ять кілька резидентних програм, доступ до кожної з яких може здійснюватися за допомогою окремої комбінації клавіш. Після виклику резидентної програми користувач взаємодіє з нею в міру необхідності, потім виконує команду TSR для повернення до запущеного в даний момент основного додатка.

Незважаючи на обмеження в багатьох відношеннях (зокрема споживання частини оперативної пам'яті максимальним об'ємом 640 Кбайт, доступне на ранніх ПК), резидентні програми фактично давали змогу запускати кілька програм у єдиному адресному просторі ОЗП.

Розробка резидентних програм була складним завданням. Більш просунуті резидентні програми, доступні в 1980-х і 1990-х роках, використовували недокументовані функції MS-DOS, щоб забезпечити максимум можливостей для своїх користувачів.

Унаслідок таких складнощів резидентні програми набули репутації таких, що спричиняють нестабільність системи. Було очевидно, що необхідний інший підхід до підтримки мультипрограмного режиму.

Використання **віртуальної пам'яті** допомогло подолати найскладніші проблеми, які перешкоджали широкому застосуванню мультипрограмного режиму в оригінальних ПК. **Віртуальна пам'ять** – це метод керування пам'яттю, що дає змогу кожному застосунку працювати у власному просторі пам'яті, зовні незалежно від будь-яких інших застосунків, що можуть одночасно перебувати в запущеному стані в одній системі. На комп'ютері із системою керування віртуальною пам'яттю операційна система відповідає за виділення фізичної пам'яті системним процесам і призначеним для користувача додаткам. Апаратні засоби та програмне забезпечення для керування пам'яттю перетворюють запити до пам'яті, які виходять із контексту віртуальної пам'яті додатка, в адреси фізичної пам'яті.

Крім спрощення процесу розроблення та запуску паралельних додатків, віртуальна пам'ять також дає змогу виділяти більший об'єм пам'яті, ніж наявний у комп'ютері. Це досягається завдяки використанню допоміжного сховища (зазвичай файлу на диску)

для тимчасового зберігання копій розділів пам'яті, що видаляються з фізичної пам'яті, щоб інша програма (або інша частина тієї самої програми) могла запуститися в пам'яті, що звільнилася. У сучасних комп'ютерах загального призначення розділи пам'яті зазвичай виділяють і переміщують у кількості, кратній фрагменту фіксованого розміру, який називають **сторінкою**. Сторінки пам'яті, як правило, мають розмір 4 Кбайт або більше.

Переміщення сторінок пам'яті в допоміжне сховище і з нього в системах віртуальної пам'яті називається **підкачуванням сторінок**. Файл, що містить замінені сторінки, є **файлом підкачки**.

У системі з керуванням віртуальною пам'яттю ні розробникам додатків, ні самому коду не потрібно піклуватися про те, скільки інших додатків запущено в системі або наскільки заповнена фізична пам'ять. У міру того, як додаток резервує пам'ять для масивів даних і виконує виклики бібліотечних підпрограм (для чого потрібне завантаження коду для цих підпрограм у пам'ять), операційна система керує розподілом фізичної пам'яті та вживає необхідних заходів, щоб кожна програма одержувала пам'ять за запитом. Тільки в малоймовірному випадку повного заповнення доступної фізичної пам'яті при одночасному досягненні граничного розміру файлу підкачки система змушена повертати код помилки у відповідь на запит про виділення пам'яті.

Крім спрощення роботи програмістів віртуальна пам'ять надає низку інших важливих переваг.

- Додатки можуть ігнорувати присутність один одного, а також не можуть втручатися, випадково або навмисно, в роботу один одного. Апаратні засоби керування віртуальною пам'яттю гарантують, що кожна програма отримує доступ тільки до тих сторінок пам'яті, які були їй призначені.

  Спроби отримати доступ до пам'яті іншого процесу або до будь-якої іншої адреси за межами виділеного застосунку простору пам'яті призводять до появи винятку, викликаного **порушенням прав доступу**.

- Кожна сторінка пам'яті має набір атрибутів, які обмежують підтримувані нею типи операцій. Сторінка може бути позначена як доступна тільки для читання, що виключає будь-які спроби запису даних у неї. Сторінка може бути позначена як виконувана – це означає, що вона містить код, який може бути виконаний у вигляді інструкцій процесора. Сторінка може бути позначена як доступна для читання і запису – такі сторінки додаток може змінювати на свій розсуд. Встановлюючи ці атрибути належним чином, операційні системи можуть підвищити стабільність системи, гарантуючи неможливість внесення змін до інструкцій процесора та виконання даних у вигляді інструкцій незалежно від того, чи є така спроба результатом випадковості або злого умислу.

- Сторінкам пам'яті може бути призначений лише мінімально необхідний рівень привілеїв, що дає змогу дозволити доступ до таких сторінок тільки коду, запущеному з привілеями ядра. Це обмеження гарантує, що операційна система продовжить працювати належним чином навіть за наявності додатків, які працюють неправильно. Завдяки цьому системну пам'ять можна відображати в адресний простір кожного процесу, забороняючи коду програми безпосередньо взаємодіяти з цією пам'яттю. Додатки можуть звертатися до системної пам'яті тільки побічно, через програмний інтерфейс, що реалізує системні виклики.
- Сторінки пам'яті можуть бути позначені для спільного використання додатками. Це означає, що до такої сторінки може бути явно дозволено доступ більш ніж з одного процесу. Це забезпечує ефективну взаємодію між процесами.

У ранніх версіях Microsoft Windows було реалізовано деякі функції віртуалізації пам'яті з використанням можливостей сегментації пам'яті процесорів 80286 і 80386. У контексті Windows використання віртуальної пам'яті остаточно оформилося з випуском Windows NT 3.1 у 1993 р. Архітектуру системи Windows NT було засновано на архітектурі **Virtual Address eXtension (VAX)**, розробленій корпорацією Digital Equipment Corporation у 1970-х роках. У цій архітектурі було реалізовано 32-розрядне середовище віртуальної пам'яті з віртуальним адресним простором об'ємом 4 Гбайт, доступним для кожного з потенційно багатьох додатків, які працюють у контексті мультипрограмного режиму. Одним з основних архітекторів операційної системи VAX (**virtual memory system, VMS**), був Девід Катлер (David Cutler), який пізніше очолив розробку Microsoft Windows NT.

Windows NT має плоску 32-розрядну організацію пам'яті. Це означає можливість прямого доступу до будь-якої адреси у всьому 32-розрядному просторі з використанням 32-розрядної адреси. Для маніпулювання сегментними регістрами не потрібно ніяких додаткових зусиль програміста. За замовчуванням віртуальний адресний простір Windows NT розділений на два блоки рівного розміру: користувацький адресний простір об'ємом 2 Гбайт у нижній половині діапазону та простір ядра об'ємом 2 Гбайт у його верхній половині.

Наступний розділ присвячено віртуальній пам'яті зі сторінковою організацією в 32-розрядній операційній системі Windows NT на процесорах Intel. Windows NT не повною мірою відображає реалізацію віртуальної пам'яті в інших операційних системах, однак у ній застосовано аналогічні принципи, а інші середовища відрізняються лише в деталях. Це введення дає уявлення про концепції віртуальної пам'яті, залишаючи додаткові подробиці, пов'язані з більш сучасними архітектурами, такими як 64-розрядні процесори та операційні системи, для наступних розділів.

Віртуальна пам'ять зі сторінковою організацією

У 32-розрядній ОС Windows NT на процесорах Intel сторінки пам'яті мають розмір 4 Кбайт. Це означає, що для адресації комірки на конкретній сторінці потрібна 12-бітна адреса (оскільки $2^{12} = 4096$). Решта 20 біт 32-розрядної віртуальної адреси використовуються в процесі перетворення (трансляції) віртуальної адреси у фізичну.

У Windows NT усі адреси пам'яті в програмі (зазначені як у вихідному коді, так і в скомпільованому виконуваному коді) є віртуальними адресами. Вони не пов'язані з фізичними адресами доти, доки програму не буде запущено під керуванням блоку керування пам'яттю.

Безперервний розділ фізичної пам'яті розміром 4 Кбайт у Windows NT називають **сторінковим кадром**. Це найменша одиниця пам'яті, керована системою віртуальної пам'яті Windows. Кожен сторінковий кадр починається з межі в 4 Кбайт. Це означає, що на початку будь-якого сторінкового кадру всі молодші 12 біт адреси дорівнюють нулю. Система відстежує інформацію, пов'язану зі сторінковими кадрами, за допомогою таблиць сторінок.

Розмір **таблиці сторінок** у Windows NT підібрано таким чином, щоб вона займала одну сторінку розміром 4 Кбайт. Кожен 4-байтовий запис у таблиці сторінок дає змогу транслювати 32-розрядну адресу з віртуального адресного простору, використовуваного програмними структурами, у фізичну адресу, необхідну для доступу до комірки пам'яті в ОЗУ або ПЗУ. Таблиця сторінок розміром 4 Кбайт містить 1024 записи трансляції адрес сторінок. Одна таблиця сторінок керує доступом до 4 Мбайт адресного простору: множимо 1024 сторінкових кадри кожної таблиці на 4 Кбайт для кожної сторінки. Процес може мати кілька пов'язаних із ним таблиць сторінок, а керування всіма цими таблицями здійснюється за допомогою каталогу таблиць сторінок. Каталог таблиць сторінок являє собою сторінку розміром 4 Кбайт, що містить серію 4-байтових посилань на таблиці сторінок.

Каталог таблиць сторінок може містити 1024 посилання на таблиці сторінок. Один каталог таблиць сторінок охоплює всі 4 Гбайт адресного простору 32-розрядної ОС Windows NT (множимо 4 Мбайт для кожної таблиці сторінок на 1024 посилання на таблиці сторінок).

Кожен процес Windows NT має окремий каталог таблиць сторінок, набір таблиць сторінок і набір сторінкових кадрів, виділених для використання цим процесом. Таблиці сторінок процесу застосовні до всіх потоків процесу, оскільки всі потоки процесу спільно використовують один і той самий адресний простір і розподіл пам'яті.

Коли системний планувальник перемикається з одного процесу на інший, контекст віртуальної пам'яті вхідного процесу замінює контекст вихідного процесу. Процесори Intel x86 підтримують адресу поточного каталогу таблиць сторінок процесу в регістрі *CR3*, також відомому як **базовий регістр каталогу сторінок** (page directory base register, PDBR). Це точка входу в каталог таблиць сторінок і таблиці сторінок, за

допомогою якої процесор може перетворити будь-яку дійсну віртуальну адресу на відповідну фізичну адресу.

Звертаючись до довільної (допустимої) комірки пам'яті та припускаючи, що інформація, яка могла б прискорити доступ, ще не збережена в кеші нещодавніх трансляцій віртуальної адреси у фізичну, процесор спочатку шукає фізичну адресу відповідної таблиці сторінок у каталозі таблиць сторінок, використовуючи старші 10 біт віртуальної адреси. Потім він звертається до потрібної таблиці сторінок і використовує наступні за значенням 10 біт адреси для вибору фізичної сторінки, що містить запитані дані. Після цього молодші 12 біт адреси вказують у сторінковому кадрі осередок пам'яті, запитаний виконуваною інструкцією.

СТОРІНКОВІ КАДРИ НЕ ВІДОБРАЖАЮТЬ ДІЙСНІ РОЗДІЛИ ФІЗИЧНОЇ ПАМ'ЯТІ



Фізична пам'ять насправді не розділена на сторінкові кадри. Структура сторінки – це лише метод, який використовується системою для відстеження інформації, необхідної для трансляції віртуальних адрес в адреси комірок фізичної пам'яті.

Для того щоб відповідати очікуванням користувачів щодо продуктивності, кожне звернення до пам'яті має бути якомога швидшим. Під час виконання кожної інструкції для вилучення кодів операцій інструкцій і даних виконується принаймні одне перетворення віртуальної адреси у фізичну. Через високу частоту повторення цього процесу розробники процесорів докладають значних зусиль для досягнення максимальної ефективності трансляції віртуальних адрес.

У сучасних процесорах у кеші трансляцій зберігаються результати недавніх пошуків для трансляції адрес віртуальної пам'яті. Такий підхід дає змогу виконувати вельми значну частку трансляцій адрес віртуальної пам'яті всередині процесора без додаткових тактових циклів, що знадобилися б, якби процесору необхідно було спочатку здійснити пошук адреси таблиці сторінок у каталозі таблиць сторінок, а потім одержати доступ до таблиці сторінок для визначення запитуваної фізичної адреси.

Структури даних, які використовуються під час трансляції віртуальної адреси у фізичну, недоступні додаткам, що працюють на рівні привілеїв користувача. Усі дії, пов'язані з трансляцією адрес, виконуються в апаратних засобах процесора і в програмних процесах, що працюють у режимі ядра.

Для того щоб прояснити процедуру трансляції віртуальних адрес, на рис. 7.4 наведено приклад трансляції 32-розрядної віртуальної адреси у фізичну в ОС Windows.

Розглянемо процес трансляції, представлений на рис. 7.4, крок за кроком. Припустімо, що процесор запитує 8-бітове значення даних, що зберігається за віртуальною

адресою *$00402003*, за допомогою інструкції *moval*, [*ebx*], при цьому в регістр *ebx* раніше було завантажено значення *$00402003*. Також припустимо, що дані трансляції цієї адреси ще не були збережені в кеші недавніх трансляцій віртуальної адреси у фізичну і сторінка перебуває в основній пам'яті. Процес трансляції описує така процедура.

1. Процесор намагається виконати інструкцію *moval*, [*ebx*], але він не може зробити це, оскільки в нього немає безпосереднього доступу до інформації, необхідної для трансляції віртуальної адреси, що зберігається в *ebx*, у відповідну фізичну адресу. Ця ситуація породжує виняток **«відмова сторінки»**, у результаті керування передається операційній системі, щоб вона могла розв'язати проблему з трансляцією адреси. У цьому разі використання терміна «відмова» не означає, що сталася якась помилка. Відмови сторінок – це звичайна складова виконання програми.

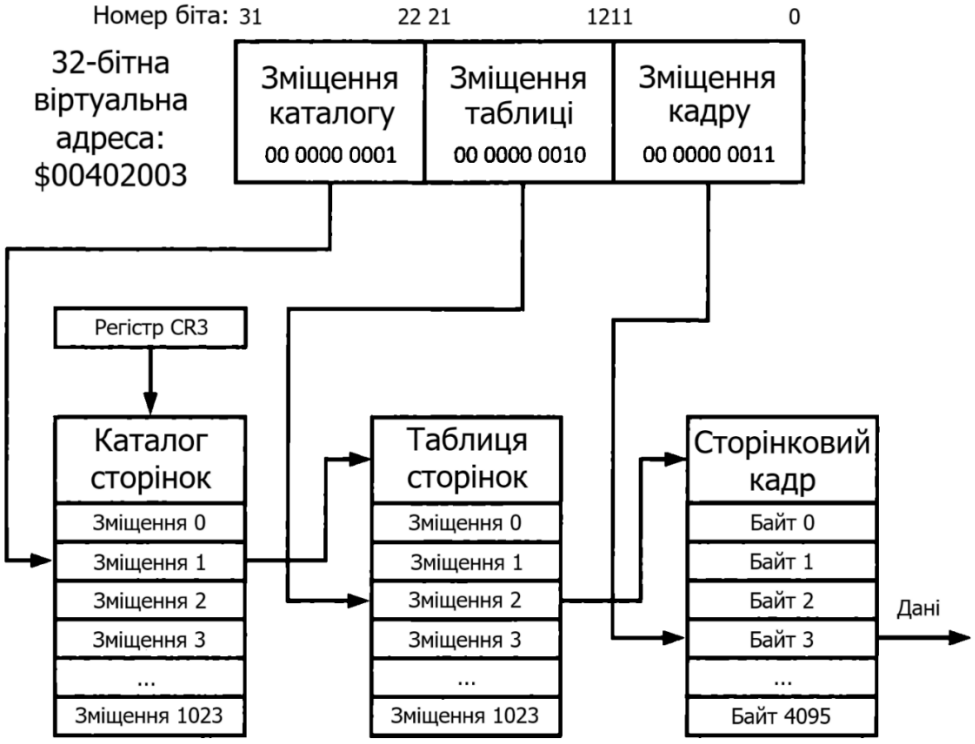


Рис 7.4. *Трансляція віртуальної адреси у фізичну*

2. Запитану віртуальну адресу зсувають праворуч на 22 бітові позиції, залишаючи 10-бітове зміщення каталогу, яке в цьому прикладі має значення *1*.
3. Зсув каталогу зсувається вліво на 2 бітові позиції (оскільки кожен запис у каталозі сторінок становить 4 байти) і додається до вмісту регістра *CR3* процесора (базовий регістр каталогу сторінок). Результатом є адреса запису в каталозі таблиць сторінок, що містить адресу відповідної таблиці сторінок.
4. Запитана віртуальна адреса зсувається вправо на 12 бітових позицій і маскується, щоб залишилося тільки 10-бітове зміщення таблиці, яке в цьому прикладі має значення *2*.

5. Зсув таблиці зсувається вліво на 2 бітові позиції (оскільки кожен запис у цій таблиці також має довжину 4 байти) і додається до адресу таблиці сторінок, зазначеної на *кроці 3*. 32-розрядна адреса, що зчитується з цього осередку, є фізичною адресою кадру сторінки, що містить запитані дані.
6. Процесор зберігає дані трансляції, яка являє собою перетворення старших 20 бітів віртуальної адреси на відповідні старші 20 бітів адреси сторінкового кадру, у своєму кеші трансляцій.
7. Процесор знову намагається виконати інструкцію *moval*, [*ebx*], яка тепер успішно переміщує запитаний байт даних у регістр *al*, використовуючи поміщені в кеш даних трансляції віртуальної адреси в фізичну. Для доступу до запитаного байта молодші 12 біт віртуальної адреси (зсув кадру), що містять значення *3* у цьому прикладі, додаються до адреси кадру сторінки, обчисленої на *кроці 5*.

Після завершення цих дій трансляція для запитаного сторінкового кадру протягом деякого часу залишається доступною в кеш-пам'яті трансляцій процесора. Поки дані трансляції залишаються в кеші, наступні запити тієї самої віртуальної адреси або інших клітинок у тому самому сторінковому кадрі будуть виконуватися без затримки доти, доки запис кеша для цього сторінкового кадру не буде перезаписано під час подальшого виконання коду.

Відмова сторінки, процедура обробки якої описана на попередніх кроках, називається **м'якою відмовою сторінки**. Вона запускає трансляцію віртуальної адреси у фізичну для сторінки, яка вже доступна процесору, але відсутня в кеші трансляцій.

Жорстка відмова сторінки виникає під час спроби доступу до сторінки, яка була перенесена в допоміжне сховище. Обробка жорсткої відмови сторінки вимагає кількох додаткових кроків, включно з резервуванням сторінкового кадру для одержання запитаного кадру, запитом сторінки з допоміжного сховища та оновленням фізичної адреси сторінки в таблиці сторінок. Оскільки жорсткі відмови сторінок пов'язані з дисковими операціями, відмови цього типу мають набагато більший вплив на продуктивність програми, ніж м'які відмови сторінок.

Процес трансляції переводить старші 20 біт віртуальної адреси у відповідні 20 біт фізичної адреси. При цьому 12 біт у кожному записі таблиці сторінок залишаються доступними для збереження інформації про стан і конфігурацію для сторінкового кадру. Використання цих бітів описано в наступному розділі.

Біти стану сторінок

У табл. 7.1 описано кожен із 12 бітів стану в 32-розрядному записі таблиці сторінок Windows NT.

Таблиця 7.1. *Біти стану сторінки*

Біт	Назва	Опис
0	Valid (Допустимість трансляції)	Значення 1 вказує, що цей запис таблиці сторінок може бути використано для трансляції. Якщо цей біт дорівнює 0, інші біти можуть мати різні значення, як визначено в операційній системі. В описах наступних бітів передбачається, що біт Valid дорівнює 1
1	Write (Запис)	Значення 1 вказує, що сторінка доступна для запису, значення 0 – сторінка доступна тільки для читання
2	Owner (Власник)	Значення 1 вказує, що сторінка перебуває в режимі користувача, значення 0 – сторінка перебуває в режимі ядра
3	Write through (Наскрізний запис)	Значення 1 вказує, що зміни на сторінці мають бути негайно збережені на диск, значення 0 – зміни сторінки зберігатимуться в ОЗУ
4	Cache disabled (Кеш вимкнено)	Значення 1 вказує, що кешування цієї сторінки вимкнено, значення 0 – кешування ввімкнено
5	Accessed (Факт доступу)	Значення 1 вказує, що сторінка була прочитана або записана, значення 0 – доступ до сторінки (будь-яким способом) не здійснювався
6	Dirty (Факт запису)	Значення 1 вказує, що сторінка була записана, значення 0 – запис не здійснювався
7	Reserved (Резерв)	Не використовується
8	Global (Глобально)	Значення 1 вказує, що це перетворення може бути застосовано до всіх процесів, значення 0 – цей запис може бути застосовано тільки до одного процесу
9	Reserved (Резерв)	Не використовується
10	Reserved (Резерв)	Не використовується
11	Reserved (Резерв)	Не використовується

Процесор використовує біти стану сторінки для збереження інформації про вміст сторінки і для керування доступом до кожної сторінки з боку системних і користувацьких процесів. Біт Owner (Власник) ідентифікує сторінку як таку, що належить ядру або користувачеві. Користувацькі процеси не можуть читати або записувати будь-які сторінки, що належать ядру. Будь-яка спроба запису на сторінку, яка позначена як доступна тільки для читання (біт Write (Запис) дорівнює 0), призводить до виключення порушення доступу.

Система використовує біти стану сторінки для максимально ефективного керування пам'яттю. Якщо біт Accessed (Факт доступу) має нульове значення, сторінка була

зарезервована, але ніколи не використовувалася. Коли системі необхідно звільнити фізичну пам'ять, сторінки, до яких ніколи не зверталися, є основними кандидатами на видалення, оскільки під час видалення з пам'яті немає необхідності зберігати їхній вміст. Аналогічно, якщо біт Dirty (Факт запису) має нульове значення, дані сторінки не змінювалися відтоді, як вони були перенесені в пам'ять. Диспетчер пам'яті може звільняти сторінки з нульовим бітом Dirty (Факт запису), оскільки відомо, що коли сторінка знову знадобиться, її можна буде знову завантажити з вихідного місця розташування (зазвичай це файл на диску), щоб точно відновити.

Сторінки зі встановленим бітом Dirty (Факт запису) мають бути збережені у файлі підкачки під час їх видалення з пам'яті. Коли сторінка переміщується у файл підкачки, відповідний запис у таблиці сторінок оновлюється з використанням іншого формату з табл. 7.1, щоб указати на неприпустимість використання цієї сторінки для перетворення (біт Valid (Допустимість трансляції) дорівнює 0) та зберегти її місцезнаходження в межах файлу підкачки. Формат записів таблиці сторінок визначається архітектурою процесора, якою в даному випадку є архітектура сімейства Intel x86. Апаратні засоби процесора звертаються до записів таблиці сторінок для трансляції віртуальної адреси у фізичну і для забезпечення захисту сторінок під час роботи процесора на повній швидкості. На додаток до керування пам'яттю, використовуваною кожним процесом, система повинна відстежувати всі сторінкові кадри ОЗУ і ПЗУ на комп'ютері незалежно від того, використовуються вони процесом чи ні. Система зберігає цю інформацію в списках, званих пулами пам'яті, які розглядаються далі.

Пупи пам'яті

У Windows NT пули пам'яті розділені на два типи: з підкачуванням і без нього.

- **Пул без підкачки** містить усі сторінкові кадри, які гарантовано завжди залишаються резидентними в пам'яті. Код для процедур обслуговування переривань, драйверів пристроїв і самого диспетчера пам'яті завжди повинен залишатися негайно доступним для процесора з міркувань підтримання необхідної продуктивності системи, а у випадку самого диспетчера пам'яті – для того, щоб система взагалі могла функціонувати. Віртуальні адреси, що не вивантажуються, знаходяться в системній частині віртуального адресного простору процесу.
- **Пул з підкачуванням** містить сторінки віртуальної пам'яті, які за потреби можуть бути тимчасово вивантажені з фізичної пам'яті.

Система відстежує стан кадру кожної сторінки у фізичній пам'яті за допомогою структури, званої базою даних **номерів сторінкових кадрів** (page frame number, PFN). PFN – це старі 20 біт фізичної базової адреси сторінкового кадру. Кадр кожної

сторінки може перебувати в одному з декількох станів залежно від його поточного та попереднього використання. Ось деякі з основних станів сторінкового кадру.

- **Active** (Активний). Сторінковий кадр є частиною робочого набору сис-темного або користувацького процесу. **Робочий набір** – це частина віртуального адресного простору процесу, присутня в даний момент у фізичній пам'яті.
- **Standby** (Той, що очікує). Сторінки, що очікують, – це сторінки, які були видалені з робочих наборів процесу і не були змінені.
- **Modified** (Змінений). Змінені сторінки були видалені з робочих наборів процесу і були змінені. Ці сторінки мають бути записані на диск, перш ніж їхні кадри можуть бути використані повторно.
- **Free** (Вільний). Вільні сторінки не використовуються, але ще містять дані про свою останню участь у робочому наборі. З міркувань безпеки ці сторінки не можуть бути доступні призначеному для користувача процесу доти, доки їхній вміст не буде замінено нулями.
- **Zeroed** (Скинутий). Скинуті сторінки є вільними, а їхній вміст було перезаписано нулями. Ці сторінки доступні для резервування користувацькими процесами.
- **Bad** (Пошкоджений). Пошкоджені сторінки викликали апаратні помилки під час звернення до них процесора. Такі сторінки відстежуються в базі даних PFN і не використовуються операційною системою.

У міру запуску, виконання і завершення роботи системних служб і додатків сторінкові кадри переходять з одного стану в інший під керуванням системи. У Windows NT системне завдання виконується в періоди простою і перетворює вільні сторінки в скинуті, перезаписуючи ці сторінки нулями.

Обговорення в цьому розділі було зосереджено на реалізації віртуальної пам'яті в архітектурі процесорів x86 під керуванням ОС Windows NT. Інші процесорні архітектури та операційні системи реалізують віртуальну пам'ять, використовуючи аналогічні концепції.

Компонент процесора, що керує розподілом пам'яті, перетворенням адрес і функціями захисту, називається **блоком керування пам'яттю**. У наступному розділі він розглядається як узагальнений компонент комп'ютерної системи.

Блок керування пам’яттю

У процесорних архітектурах, що підтримують віртуальну пам'ять зі сторінковою організацією, функції **блоку керування пам'яттю** (memory management unit, MMU) реалізуються або всередині самого процесора, або іноді (особливо в досить старих

рішеннях) у вигляді окремої інтегральної схеми. У MMU віртуальний адресний простір процесора розділено на одиниці розподілу пам'яті розміром зі сторінку.

Сторінки можуть мати фіксований розмір, як у випадку Windows NT, або MMU може підтримувати кілька розмірів. Сучасні процесори, включно з процесорами x86 пізніших поколінь, часто підтримують сторінки двох розмірів: малі та великі. Малі сторінки зазвичай мають розмір кілька кілобайтів, а велика сторінка може займати кілька мегабайтів. Підтримка великих сторінок дає змогу уникнути неефективності, пов'язаної з виділенням безлічі сторінок меншого розміру під час роботи з великими об'єктами даних.

Як обговорювалося раніше, MMU зазвичай містить кеш для підвищення швидкості доступу до пам'яті завдяки виключенню необхідності переглядати каталог таблиць сторінок і виконувати пошук у таблиці сторінок під час кожного звернення до пам'яті. Використання кешування для підвищення продуктивності є темою *розділу 8*, однак тут ми представимо кеш для трансляції віртуальних адрес у фізичні, оскільки ця конструкція є основною особливістю більшості архітектур MMU.

Компонент кешування в MMU, який зберігає дані раніше виконаних трансляцій віртуальних адрес у фізичні, називається **буфером асоційованої трансляції** (translation lookaside buffer, TLB). Для того щоб уникнути пошуку таблиці сторінок у каталозі таблиць сторінок, а потім пошуку в знайденій таблиці сторінок кадру потрібної сторінки під час кожного звертання до пам'яті, TLB зберігає дані про трансляцію віртуальних адрес у сторінкові кадри, отримані в результаті цих пошуків, в апаратній структурі, яка називається **асоціативною пам'яттю**.

Щоразу, коли процесору потрібно отримати доступ до фізичної пам'яті, що може відбуватися кілька разів за час виконання однієї інструкції, він спочатку перевіряє асоціативну пам'ять TLB, щоб визначити, чи немає в TLB інформації про потрібну трансляцію. Якщо така інформація в буфері є, інструкція негайно використовує її для доступу до фізичної пам'яті. Якщо TLB не містить даних запитаної трансляції, виникає відмова сторінки, і процесор повинен заново переглянути каталог таблиць сторінок і таблицю сторінок, щоб визначити трансльовану адресу, припускаючи, що потрібна сторінка знаходиться в пам'яті.

На рис. 7.5 представлено роботу буфера TLB.

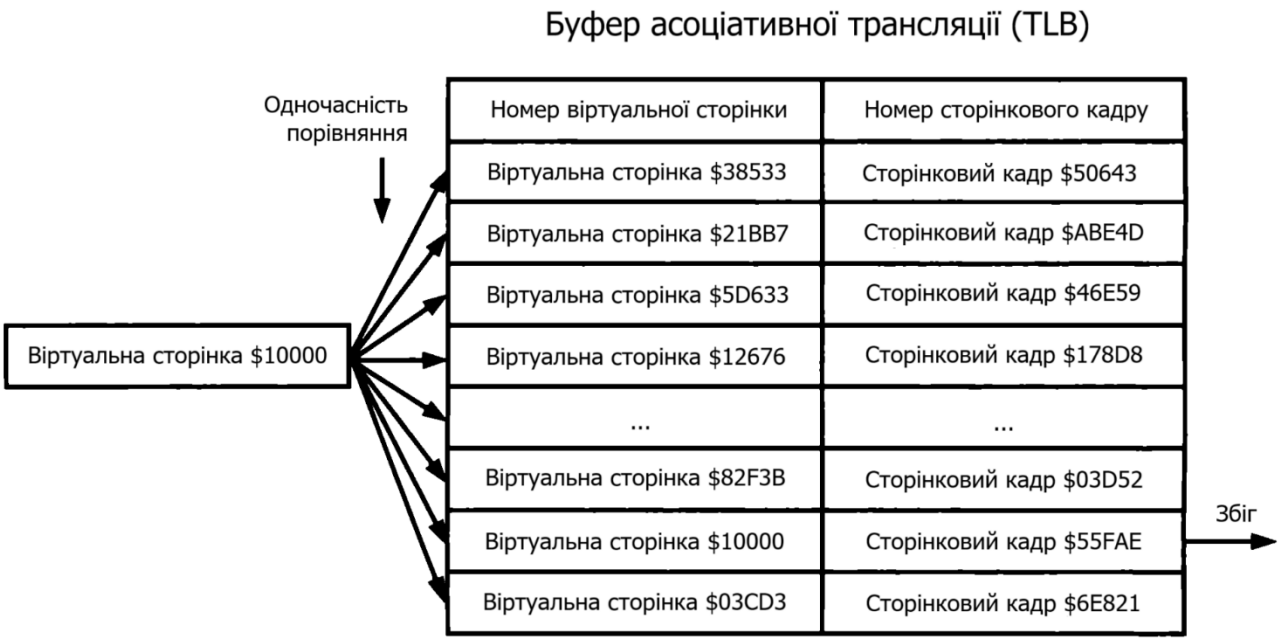


Рис 7.5. *Робота буфера асоціативної трансляції*

При кожному зверненні до пам'яті процесор витягує з віртуальної адреси старші 20 біт, щоб ідентифікувати номер віртуальної сторінки. Цей номер сторінки, *$10000*, у цьому прикладі використовується для пошуку в TLB відповідного запису. Апаратні засоби TLB одночасно порівнюють номер запитаної віртуальної сторінки з номерами всіх віртуальних сторінок, що зберігаються в TLB. Якщо знайдено збіг, номер кадру відповідної сторінки негайно надається для отримання доступу до фізичної пам'яті.

TLB містить обмежену кількість записів, зазвичай 64 або менше. Процесор повинен керувати тим, які записи TLB зберігаються, а які відкидаються в міру опрацювання запитів адрес для різних комірок пам'яті. Коли буфер TLB заповнений, MMU вирішує, який з наявних записів TLB перезаписати новою інформацією. MMU може випадковим чином вибрати запис TLB для заміни, більш складна реалізація MMU може використовувати інформацію про термін існування записів для заміни запису TLB, який найдовше не використовувався.

На додаток до трансляції віртуальної адреси у фізичну MMU зазвичай виконують такі функції.

- **Поділ віртуальної пам'яті на простір ядра і простір користувача.** Пам'ять ядра зарезервована для використання операційною системою і пов'язаними з нею компонентами, такими як драйвери пристроїв. Простір користувача доступний для додатків і для дій, ініційованих користувачами, наприклад для обробки команд, набраних у вікні командного рядка. Код користувацького рівня не може отримати прямий доступ до системної пам'яті. Замість цього код користувача повинен викликати системні функції для запиту таких послуг, як виділення пам'яті та операції введення-виведення.
- **Ізоляція областей пам'яті процесів.** Кожен процес має власний адресний простір, що є єдиною областю пам'яті, до якої йому дозволено доступ. Якщо

між процесами не встановлена дозволена системою область пам'яті спільного використання, будь-якому процесу заборонено звертатися до пам'яті, використовуваної іншим процесом. Один процес не може помилково або навмисно змінити вміст пам'яті, яка належить до приватної області пам'яті для іншого процесу.

- **Обмеження доступу на рівні сторінок.** На додаток до захисту системних сторінок від доступу користувачів і захисту приватних сторінок процесів від доступу інших процесів процес може встановлювати захист на окремі сторінки, якими він володіє. Сторінки можуть бути позначені як доступні тільки для читання, що забороняє зміну їхнього вмісту. Сторінки, позначені як заборонені для виконання, не можуть бути використані для надання інструкцій, що виконуються процесором. У деяких архітектурах сторінки можуть бути позначені як заборонені для доступу, що встановлює заборону як на читання, так і на запис. Сторінки, що містять виконуваний код, можуть бути позначені як доступні тільки для читання, щоб запобігти випадковій або навмисній зміні інструкцій у пам'яті.
- **Виявлення проблем із програмним забезпеченням.** У деяких мовах програмування, зокрема в мові C, на жаль, часто спостерігаються спроби використати покажчик, що містить неприпустиму адресу (**покажчик** – це змінна, що містить адресу іншої змінної). Найчастіше в такій ситуації трапляється неприпустима адреса 0, оскільки змінні нерідко ініціалізуються нулем. Ця проблема настільки поширена, що реакція системи на неї має власну назву – **виключення нульового покажчика**. Коли програма мовою C намагається отримати доступ до комірки пам'яті, що не входить у допустимий діапазон віртуальних адрес програми, наприклад до адреси $00000000, MMU спричиняє виняток, який, якщо його не опрацювати в програмі, зазвичай призводить до аварійного завершення програми з виведенням повідомлення про помилку у вікно консолі. У системах без віртуальної пам'яті звернення до помилкових осередків можуть просто призводити до зчитування або запису пам'яті за вказаною адресою без будь-якої індикації помилки, що призводить до некоректної роботи застосунку або всієї системи. Такі помилки в системах без MMU можуть бути вкрай складними для виправлення, якщо проблема проявляється не відразу.

Сучасні процесори під керуванням Linux, Windows і більшості операційних систем для мобільних пристроїв зазвичай вимагають, щоб їхні хост системи використовували керування віртуальною пам'яттю та забезпечували механізми захисту сторінок, описані в цьому розділі.

Вбудовані процесори реального часу, що виконують критично важливі для безпеки завдання, такі як керування польотом авіалайнера або роботою автомобільної

подушки безпеки, можуть підтримувати або не підтримувати повний набір функцій MMU. Одним із недоліків, пов'язаних із використанням віртуальної пам'яті в системах жорсткого реального часу, є змінна часова затримка, що виникає внаслідок необхідності опрацювання м'яких відмов сторінок, а також, якщо реалізовано підкачування сторінок, то й жорстких відмов сторінок. Оскільки в багатьох ОСРЧ час виконання має суворо контролюватися, їх розробники часто уникають використання віртуальної пам'яті. Такі системи не містять MMU, але в них часто реалізують багато інших можливостей MMU, як-от апаратний захист системної пам'яті та керування доступом до областей оперативної пам'яті.

Резюме

У цьому розділі було розглянуто основні категорії архітектур сучасних процесорів, включно з фон-Нейманівською, гарвардською та модифікованою гарвардською, а також їхнє використання в різних галузях обчислень. Було обговорено концепції віртуальної пам'яті зі сторінковою організацією, включаючи деякі деталі, що стосуються реалізації такої пам'яті в Windows NT на процесорі x86.

Розглянуто загальну структуру блока керування пам'яттю (MMU) з акцентом на використання буфера асоціативної трансляції (TLB) як методу оптимізації продуктивності під час трансляції віртуальних адрес у фізичні.

У наступному розділі ми не обмежимося підвищенням продуктивності, забезпечуваним TLB, і докладно розглянемо широко використовувані методи прискорення процесорів, включно з кешуванням, конвеєрним опрацюванням інструкцій і паралелізмом інструкцій.

Вправи

1. 16-розрядний вбудований процесор має окремі області пам'яті для коду і даних. Код зберігається у флеш-пам'яті, а змінювані дані – в оперативній пам'яті. Деякі значення даних, такі як константи і початкові значення для елементів даних оперативної пам'яті, зберігаються в тій самій області флеш-пам'яті, що й інструкції програми. ОЗУ і ПЗУ знаходяться в одному адресному просторі. Яка з архітектур процесорів, розглянутих у цьому розділі, найкраще описує цей процесор?
2. Процесор, описаний у *вправі 1*, має функції захисту пам'яті, які не дозволяють виконуваному коду змінювати пам'ять інструкцій програми. Для доступу до інструкцій і даних цей процесор використовує фізичні адреси. Чи містить він блок керування пам'яттю (MMU)?
3. Порядок доступу до послідовних елементів у великій структурі даних може справити відчутний вплив на швидкість обробки через такі фактори, як повторне використання записів буфера TLB. Послідовний доступ до віддалених

елементів масиву (це елементи, які перебувають за межами сторінкового кадру, де розміщені раніше запитані елементи) вимагає годинних м'яких відмов сторінок у міру завантаження нових записів TLB і видалення старих.

Напишіть програму, яка створює великий двовимірний масив чисел, наприклад на 10 000 рядків і 10 000 стовпців. Виконайте ітеративний обхід масиву в порядку зростання стовпців, підписуючи кожний елемент сумою індексів рядка і стовпця. Доступ за стовпчиками означає, що індекс стовпця (другий індекс) збільшується найшвидше. Іншими словами, індекс стовпця збільшується у внутрішньому циклі.

Точно виміряйте час, який займає ця процедура. Зверніть увагу, що вам може знадобитися вжити заходів для того, щоб ваша мова програмування не виключила при оптимізації весь цей розрахунок через те, що результати роботи з масивом не використовуються надалі. Може бути достатньо вивести одне зі значень масиву після завершення відліку часу або ж може знадобитися зробити щось на кшталт підсумовування всіх елементів масиву і виведення цього результату.

Повторіть процес, включно з вимірюванням часу, точно так само, як було описано раніше, тільки змініть внутрішній цикл на ітерацію за індексом рядка (перший індекс), а зовнішній цикл – на ітерацію за індексом стовпчика, зробивши послідовність доступу рядковою.

Під час виконання вашого коду комп'ютери загального призначення виконують безліч інших завдань, тому, щоб отримати статистично достовірний результат, вам може знадобитися виконати обидві процедури кілька разів. Для початку можна провести експеримент 10 разів і усереднити час доступу до масиву за стовпчиками і за рядками.

Чи можете ви визначити метод доступу до масиву, який постійно дає найкращий результат? Який порядок є найшвидшим у вашій системі при використанні обраної вами мови? Слід враховувати, що різниця між методами з порядком доступу за стовпчиками і за рядками може бути не дуже значною – вона може становити всього кілька відсотків.

8

Методи підвищення продуктивності

Фундаментальні аспекти архітектур процесорів і пам'яті, розглянуті в попередніх розділах, слугують основою для повнофункціональної комп'ютерної системи. Однак продуктивність такої системи порівняно з більшістю сучасних процесорів буде невисокою, якщо в неї не додати функції, що дають змогу збільшити швидкість виконання інструкцій.

Під час проектування процесорів і систем зазвичай використовують кілька методів підвищення продуктивності для досягнення максимальної швидкості виконання в реальних комп'ютерних системах. Ці методи не змінюють те, що робить процесор з погляду виконання програм і обробки даних; вони просто допомагають робити це швидше.

Після прочитання цього розділу ви зрозумієте вигоду від застосування багаторівневої кеш-пам'яті в комп'ютерних архітектурах, а також орієнтуватиметеся в перевагах і проблемах, пов'язаних із конвеєрним опрацюванням інструкцій. Ви також дізнаєтеся, як підвищити продуктивність за допомогою одночасної багатопотоковості, і ознайомитеся з призначенням і застосуванням моделі обробки з одним потоком інструкцій і безліччю потоків даних (single instruction, mul-tiple data, SIMD).

У цьому розділі буде розглянуто такі теми:

- кеш-пам'ять;
- конвеєрна обробка інструкцій;
- одночасна багатопотоковість;
- модель обробки SIMD.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою https://github.com/PacktPublishing/Modern-Computer-Architecture-andOrganizайon-Second-Edition.

Кеш-пам’ять

Кеш-пам’ять – це область високошвидкісної пам'яті (порівняно зі швидкістю основної пам'яті), у якій тимчасово зберігаються програмні інструкції або дані для майбутнього використання. Як правило, ці інструкції або елементи даних були нещодавно витягнуті з основної пам'яті і, ймовірно, незабаром знову знадобляться.

Основне призначення кеш-пам'яті – збільшити швидкість повторного доступу до однієї і тієї самої комірки пам'яті, а також до комірок пам'яті, розташованих поруч із нею. Для того щоб забезпечити ефективне прискорення, доступ до кешованих елементів має бути значно швидшим, ніж доступ до вихідного джерела інструкцій або даних, званого **резервним сховищем**.

Коли використовується кешування, кожна спроба доступу до комірки пам'яті починається з пошуку в кеші. Якщо запитаний елемент присутній у кеші, процесор негайно витягує його і використовує. Це називається **кеш-попаданням.** Якщо пошук у кеші не увінчався успіхом **(кеш-промах)**, інструкцію або елемент даних витягують із резервного сховища. У процесі вилучення запитаного елемента його копія додається в кеш для передбачуваного використання в майбутньому.

Кеш-пам'ять використовується в комп'ютерних системах для різних цілей. Нижче наведено деякі приклади застосування кеш-пам'яті.

- **Буфер асоціативної трансляції (translation lookaside bupep, TLB).** TLB, як було показано в *розділі 7*, являє собою варіант кеш-пам'яті, яка використовується в процесорах, що підтримують віртуальну пам'ять зі сторінковою організацією. TLB містить набір записів про трансляцію віртуальних адрес у фізичні, які прискорюють доступ до сторінкових кадрів у фізичній пам'яті. У міру виконання інструкцій кожне звернення до основної пам'яті вимагає трансляції віртуальної адреси у фізичну. Швидкий пошук у TLB призводить до набагато швидшого виконання інструкцій порівняно з процесом пошуку в таблицях сторінок після промаху TLB. Буфер TLB є частиною блоку керування пам'яттю (MMU) і не має прямого відношення до варіантів кеш-пам'яті процесора, що обговорюються далі в цьому розділі.
- **Кеш-пам'ять дискових накопичувачів.** Операції читання і запису на намагнічені пластини дискових накопичувачів виконуються на порядки повільніше, ніж доступ до пристроїв **динамічної пам'яті довільного доступу** (dynamic RAM, DRAM). У дискових накопичувачах кеш-пам'ять зазвичай

використовується для зберігання вихідних даних операцій читання і тимчасового зберігання даних при підготовці до запису. Контролери накопичувачів часто зберігають у внутрішній кеш-пам'яті більше даних, ніж спочатку запитану кількість, розраховуючи на те, що під час майбутніх звернень будуть запитуватися дані, що зберігаються поруч із спочатку запитаними. Якщо це припущення виявиться правильним, як часто буває, накопичувач може негайно відреагувати на другий запит, передавши дані з кеша без затримки, пов'язаної з доступом до дисків.

- **Кеш-пам'ять веб-браузерів.** Веб-браузери зазвичай зберігають копії нещодавно відвіданих веб-сторінок у пам'яті в очікуванні натискання користувачем кнопки **Back** (Назад) для повернення на раніше переглянуту сторінку. Коли це відбувається, браузер може витягти частину або весь вміст цієї сторінки зі свого локального кешай одразу ж знову відобразити сторінку без необхідності доступу до віддаленого веб-сервера і повторного отримання тієї самої інформації.
- **Кеш-пам'ять інструкцій і даних процесора.** Структури кеш-пам'яті процесора розглядаються в наступних розділах. Мета цієї кеш-пам'яті – підвищити швидкість доступу до інструкцій і даних завдяки усуненню затримки, що виникає під час доступу до модулів DRAM.

Кеш-пам'ять підвищує продуктивність комп'ютера, оскільки багато алгоритмів, що виконуються операційними системами і додатками, демонструють локальність посилань. Термін **"локальність посилань"** відноситься до повторного використання даних, доступ до яких було отримано нещодавно (**тимчасова локальність**), а також до повторного використання даних, що знаходяться у фізичній близькості від даних, запитаних раніше (**просторова локальність**).

Якщо взяти використання структури TLB як приклад, то тимчасова локальність застосовується в ньому шляхом збереження даних трансляцій віртуальних адрес у фізичні протягом деякого часу після початкового доступу до певного сторінкового кадру. Будь-які додаткові звернення до того самого сторінкового кадру в наступних інструкціях забезпечать набагато швидший доступ до даних трансляції, доки ці дані зрештою не будуть замінені в кеші даними іншої трансляції.

Просторову локальність TLB використовує, посилаючись на весь сторінковий кадр за допомогою одного свого запису. Будь-які наступні звернення до різних адрес на тій самій сторінці виграють за швидкістю від наявності запису TLB, отриманого в результаті першого посилання на цю сторінку.

Як правило, розмір областей кеш-пам'яті невеликий порівняно з розміром резервного сховища. Пристрої кеш-пам'яті розраховані на максимальну швидкість, що зазвичай означає, що вони є складнішими та дорожчими з розрахунку на біт, порівнюючи з

технологіями зберігання даних, що використовуються в резервному сховищі. З огляду на свій обмежений розмір пристрої кеш-пам'яті зазвичай швидко заповнюються.

Якщо в кеші немає доступного місця для зберігання нового запису, необхідно відкинути більш ранній запис. Для вибору запису кеша, який буде перезаписано новим записом, контролер кеша використовує **політику заміщення вмісту кеша**.

Мета кеш-пам'яті процесора – довести частку кеш-влучень із плином часу до максимуму, забезпечивши тим самим максимально високу середню швидкість виконання інструкцій. Для досягнення цієї мети логіка кешування повинна визначати, які інструкції і дані будуть поміщені в кеш і збережені для майбутнього використання.

Логіка кешування процесора не гарантує, що кешований елемент даних коли-небудь буде використовуватися знову після того, як його було додано в кеш.

Ця логіка ґрунтується на тому, що завдяки часовій і просторовій локальності існує досить висока ймовірність того, що звернення до кешованих даних відбудеться в найближчому майбутньому. На практиці в сучасних процесорах кеш-попадання зазвичай відбуваються в 95-97% випадків звернень до пам'яті. Оскільки затримка кеш-пам’яті становить лише малу частку від затримки DRAM, висока частота кеш-влучень призводить до істотного підвищення продуктивності порівняно з архітектурою без кеш-пам'яті.

У наступних розділах обговорюються технології багаторівневого кешування сучасних процесорів і деякі політики заміщення вмісту кеш-пам'яті, які використовуються під час їх реалізації.

Багаторівневе кешування в процесорах

За роки, що минули з моменту появи персональних комп'ютерів, процесори продемонстрували значне збільшення швидкості обробки інструкцій. Внутрішній годинник сучасних процесорів Intel і AMD майже в 1000 разів швидший, ніж у процесора 8088, що використовувався в першому IBM PC. Для порівняння, швидкість технології пам'яті DRAM з часом збільшувалася набагато повільніше. Виходячи з цих тенденцій, можна сказати, що якби сучасний процесор мав прямий доступ до DRAM для всіх своїх інструкцій і даних, він витрачав би суттєву частку свого часу на очікування відповіді DRAM на кожен запит.

Для того щоб дати для цієї теми деякі приблизні цифри, розглянемо сучасний процесор, здатний обробляти 32-бітове значення даних з регістра процесора за 1 нс. Доступ до такого значення з DRAM може зайняти 100 нс. Дещо спростимо ситуацію: якщо для кожної інструкції потрібне одне звернення до комірки пам'яті, а час виконання кожної інструкції набагато менший за час доступу до пам'яті, можна очікувати, що цикл оброблення, у якому виконується доступ до потрібних даних із

регістрів процесора, працюватиме в 100 разів швидше, ніж той самий алгоритм, який звертається до основної пам'яті при виконанні кожної інструкції.

Тепер припустимо, що в систему додано кеш-пам'ять із часом доступу 4 нс. Завдяки перевагам кеш-пам'яті алгоритм, який звертається до DRAM при виконанні кожної інструкції, буде відчувати зниження продуктивності, витрачаючи на перше звернення до певної адреси 100 нс, але наступні звернення до тих самих і сусідніх адрес будуть відбуватися зі швидкістю доступу до кеш-пам'яті – 4 нс. Доступ до кешавчетверо повільніший, ніж доступ до регістрів, однак він усе ж у 25 разів швидший за доступ до DRAM. Цей приклад показує ступінь прискорення виконання, досяжну за рахунок ефективного застосування кеш-пам'яті в сучасних процесорах.

Високопродуктивні процесори зазвичай використовують кілька рівнів кешування з метою досягнення максимальної швидкості виконання інструкцій. Апаратні засоби кеша процесора обмежені за розміром і продуктивністю економікою напівпровідникових технологій. Вибір оптимального поєднання типів і розмірів кеш-пам'яті процесора за умови отримання прийнятної для кінцевих користувачів ціни є ключовою метою розробників процесорів.

Як основна пам'ять і для внутрішнього зберігання даних процесора зазвичай використовуються схеми оперативної пам'яті двох типів – динамічна (DRAM) і статична (SRAM) оперативна пам'ять. Схеми DRAM мають невисоку ціну, але пропонують порівняно повільний доступ, здебільшого через витрати часу на заряджання та розряджання конденсаторів бітових комірок під час операцій читання та запису.

SRAM набагато швидша за DRAM, але й набагато дорожча за неї, чим зумовлено використання невеликих обсягів статичної пам'яті в системах, де продуктивність має вирішальне значення. Конструкції схем DRAM оптимізовані за щільністю, результатом чого є зберігання максимально можливої кількості бітів на одній інтегральній схемі DRAM. Конструкції схем SRAM оптимізовані за швидкістю, щоб звести до мінімуму час читання або запису комірки пам'яті. Кеш-пам'ять процесора зазвичай реалізується з використанням статичної оперативної пам'яті.

Статична оперативна пам'ять

Статична оперативна пам'ять (static RAM, SRAM) пропонує значно швидший доступ до даних порівняно з DRAM, хоча й за рахунок значно складнішої схеми. Бітові комірки SRAM займають набагато більше місця на кристалі інтегральної схеми, ніж комірки DRAM, здатні зберігати еквівалентний обсяг даних. Як ви пам'ятаєте з *розділу 4*, один бітовий осередок DRAM складається всього з одного польового МОН-транзистора і одного конденсатора.

Стандартна схема для одного біта SRAM містить шість польових МОН-транзисторів. Чотири з цих транзисторів використовуються для формування двох елементів НЕ. Ці

вентилі засновані на КМОН-схемі, показаній на рис. 4.3 *у розділі "Знайомство з польовими МОН-транзисторами" глави 4*. Ці вентилі позначені як G_1 і G_2 на рис. 8.1.

Вихід кожного елемента НЕ з'єднаний із входом іншого, утворюючи тригер. Більшу частину часу на шині слів зберігається низький рівень, що підтримує транзисторні перемикачі T_1 і T_2 у вимкненому стані, ізолюючи цю пару вентилів. Поки на шині слів діє низький рівень (і подача живлення триває), вентилі перебуватимуть в одному з двох станів:

- **збережений біт дорівнює 0**: на вході G_1 – низький рівень, а на його виході – високий;
- **збережений біт дорівнює 1**: на вході G_1 – високий рівень, а на його виході – низький.

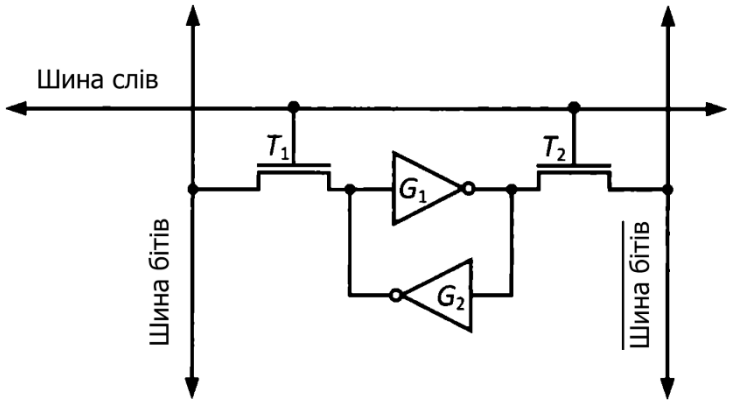


Рис 8.1. *Схема комірки SRAM*

Вхідні транзистори (T_1 і T_2) функціонують як перемикачі, які з'єднують шину бітів з коміркою для читання і запису. Як і у випадку з DRAM, переведення шини слів на високий рівень забезпечує доступ до бітового осередку за рахунок зменшення опору на кожному вхідному транзисторі до дуже малого значення. Для того щоб зчитати вміст комірки, схема зчитування вимірює напругу між парою шини бітів, позначеними Шина бітів і $\overline{\text{Шина бітів}}$ (де лінія зверху означає інверсію – операцію НЕ). Сигнали цих двох шин завжди мають протилежні значення, утворюючи диференціальну пару. Вимірювання знака різниці напруг між двома сигналами дає змогу визначити вміст комірки: 0 або 1.

Під час запису в бітову комірку шина слів переводиться на високий рівень, а на шини бітів подають сигнали з протилежними рівнями напруги, що представляють бажане значення (0 або 1) для запису. Транзистори, що записують дані в шини бітів, повинні володіти істотно більшою потужністю, ніж транзистори вентилів НЕ бітового осередку. Завдяки цьому бажане значення можна записати в осередок, навіть якщо при цьому необхідно придушити стан тригера, щоб переключити його в записуваний стан.

Банк пам'яті SRAM являє собою прямокутну сітку рядків і стовпців, як і в DRAM. Шина слів забезпечує доступ до всіх бітових комірок SRAM в одному рядку. Шини бітів з'єднуються з усіма стовпцями сітки бітових комірок.

На відміну від DRAM, в SRAM не потрібне періодичне оновлення для збереження даних, що зберігаються в комірках. Саме з цієї причини цей варіант пам'яті називають статичною оперативною пам'яттю.

У наступному розділі ми побачимо, як за допомогою SRAM створюється перший рівень кеш-пам'яті процесора.

Кеш першого рівня

У кеш-пам'яті з багаторівневою архітектурою рівні нумеруються, починаючи з 1. Кеш першого рівня (який також називають **кеш L1**) – це перша ділянка кеш-пам'яті, до якої процесор звертається під час запиту інформації або елемента даних з пам'яті. Оскільки це перше звернення до кеш-пам'яті, кеш L1 зазвичай створюють із використанням найшвидшої доступної технології SRAM, а фізично він розташований максимально близько до логічної схеми процесора.

Акцент на швидкості робить кеш L1 дорогим і енерговитратним. Це означає, що він має бути досить невеликим порівняно з основною пам'яттю, особливо в рішеннях, де витрати є важливим фактором. Навіть при скромних розмірах швидкий кеш першого рівня може забезпечити істотне підвищення продуктивності порівняно з аналогічним процесором, у якому не використовується кешування.

Процесор (або блок керування пам'яттю, MMU, якщо він присутній) переміщає дані між DRAM і кешем у вигляді блоків даних фіксованого розміру, званих **рядками кеша**. У комп'ютерах, оснащених модулями DDR DRAM, розмір рядка кеша зазвичай дорівнює 64 байтам. Такий самий розмір рядка кеша зазвичай використовується на всіх рівнях кеш-пам'яті.

У сучасних процесорах кеш L1 часто поділяють на дві секції однакового розміру: одну для інструкцій і одну для даних. Таку конфігурацію називають **розділеною кеш-пам’яттю**. У ній кеш інструкцій першого рівня називають **I-кеш L1**, а кеш даних першого рівня – **D-кеш L1**. Процесор використовує окремі шини для доступу до кожного з цих кешів, тим самим реалізуючи важливий аспект гарвардської архітектури. Така організація прискорює виконання інструкцій за рахунок паралельного доступу до інструкцій і даних з урахуванням того, що кеш L1 використовується для обох видів доступу.

У сучасних процесорах застосовують безліч стратегій для організації кеш-пам’яті та керуванні її роботою. Найпростішою конфігурацією кеш-пам'яті є кеш із прямим відображенням, представлений у наступному розділі.

Кеш із прямим відображенням

Кеш із прямим відображенням – це блок пам'яті, організований у вигляді одновимірного масиву наборів кеша, де кожна адреса в основній пам'яті зіставлена з одним набором у кеші. Кожен **набір кеша** складається з таких елементів:

- рядок кеша, що містить блок даних, зчитаних з основної пам'яті;
- значення теге, що вказує місце розташування в основній пам'яті, яке відповідає кешованим даним;
- біт допустимості, що вказує, чи містить цей набір кеша дані,

У цьому прикладі представлено кеш інструкцій, доступний тільки для читання. Кеші даних для читання і запису ми розглянемо в наступному розділі.

Бувають випадки, коли кеш не містить даних, наприклад одразу після увімкнення процесора. Біт допустимості кожного набору кеша спочатку має нульове значення, що вказує на відсутність даних у наборі. Коли біт допустимості дорівнює 0, використання цього набору для пошуку заборонено. Під час завантаження даних у набір кеша апаратні засоби встановлюють біт допустимості.

Як приклад ми будемо використовувати невеликий I-кеш L1 розміром 512 байт. Оскільки це кеш інструкцій, доступний тільки для читання, для нього не потрібна підтримка можливості запису в пам'ять. Розмір рядка кеша становить 64 байти. Ділення 512 байт на 64 байти для кожного набору дає 8 наборів кеша.

64 байти в кожному наборі – це 2^6 байт, тобто 6 найменш значущих бітів адреси визначають місце розташування в рядку кеша. Для вибору одного з восьми наборів кеша потрібні три додаткові біти адреси.

Виходячи з цієї інформації, на рис. 8.2 показано поділ 32-розрядної адреси фізичної пам'яті на компоненти: тег, номер набору і байтовий зсув рядка кеша.

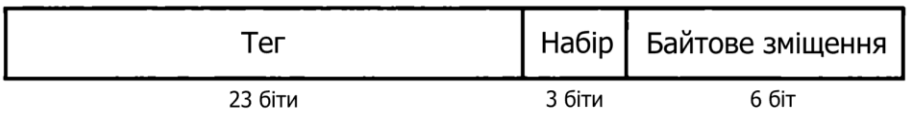


Рис 8.2. *Компоненти 32-розрядної адреси фізичної пам'яті*

Щоразу, коли процесор зчитує інструкцію з DRAM (що необхідно щоразу, коли інструкція відсутня в кеші), MMU зчитує 64-байтовий блок, який містить адресований осередок, і зберігає його в наборі I-кеша L1, обраному трьома бітами компонента "Набір", показаного на рис. 8.2. Старші 23 біти адреси зберігаються в полі "Тег" набору кеша, а біт **допустимості** встановлено.

У міру того, як процесор витягує кожну наступну інструкцію, пристрій керування використовує три біти компонента "Набір" в адресі інструкції, щоб вибрати набір кеша для порівняння. Апаратні засоби порівнюють старші 23 біти адреси виконуваної

інструкції зі значенням тегу, що зберігається у вибраному наборі кеша. Якщо ці значення збігаються, то відбулося кеш-попадання, і процесор зчитує інструкцію з рядка кешу. Якщо відбувається кеш-промах, MMU зчитує рядок із DRAM у відповідний набір кеша(перезаписуючи будь-які наявні дані для цього набору) і передає інструкцію пристрою керування для виконання.

На рис. 8.3 показано організацію всього 512-байтового кеша і його зв'язок із трьома полями в 32-розрядній адресі інструкції.

Для того щоб продемонструвати, чому кеш із прямим відображенням може забезпечити високу частоту влучань, припустімо, що ми запускаємо програму, яка містить цикл, що починається з адреси фізичної пам'яті 8000h (тут для простоти ми ігноруємо старші 16 біт 32-розрядної адреси) і містить 256 байт коду. У цьому циклі інструкції виконуються послідовно від початку до кінця 256-байтового діапазону, після чого здійснюється повернення до початку циклу для наступної ітерації.

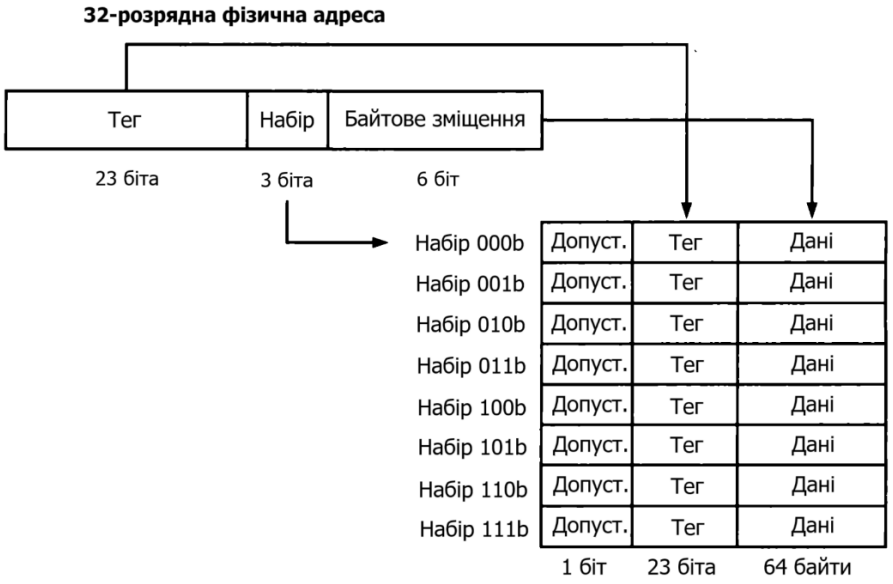


Рис 8.3. *Зв'язок 32-розрядної фізичної адреси з кеш-пам'яттю*

Адреса 8000h містить 000b у полі "Набір", тому цю адресу зіставляють із першим набором кеша, як показано на рис. 8.3. Під час першого проходу через цикл MMU витягує з DRAM 64-байтовий рядок кеша і зберігає його в першому наборі кеша (зі значенням поля "Набір", що дорівнює 000b). У міру виконання інших інструкцій, що зберігаються в тому самому 64-байтовому блоці, кожна з них буде витягуватися безпосередньо з кеша.

При переході процесу виконання до другого 64-байтового блоку знадобиться ще одне звернення до DRAM для зчитування даних. До моменту досягнення кінця циклу набори з 000b по 011b виявляються заповненими 256 байтами коду для цього циклу. Для решти проходів через цей цикл, з урахуванням припущення, що потік виконується без переривань, процесор досягне 100% кеш-потраплянь і максимальної швидкості виконання інструкцій.

З іншого боку, якщо інструкції в циклі споживають значно більше пам'яті, то вигода від кешування зменшиться. Припустимо, що інструкції циклу займають 1024 байти, тобто вдвічі більші за розмір кеша. У циклі, з його початку і до кінця, виконується одна й та сама послідовність. Коли адреси інструкцій досягають середини циклу, кеш виявляється заповненим першими 512 байтами інструкцій. На початку наступного рядка кеша після середньої точки адреса буде дорівнювати 8000h плюс 512, що дорівнює 8200h. Група бітів "Набір" в адресі 8200h – та сама, що і в адресі 8000h, що призводить до перезапису рядка кеша для адреси 8000h рядком кеша для адреси 8200h. Кожен наступний рядок кеша буде перезаписуватися в міру виконання коду другої половини циклу.

Незважаючи на те що всі області кешованої пам'яті перезаписуються під час кожного проходження циклу, структура кеш-пам'яті, як і раніше, забезпечує значну вигоду, тому що після зчитування з DRAM кожний 64-байтовий рядок залишається в кеші й доступний для використання в міру виконання інструкцій. Недоліком у цьому прикладі є підвищена частота промахів кеша. Це тягне за собою істотне зниження продуктивності, оскільки, як ми бачили, доступ до інструкції в DRAM може бути в 25 (або більше) разів повільнішим, ніж доступ до тієї ж інструкції в I-кеші L1.

ТЕГИ ВІРТУАЛЬНИХ І ФІЗИЧНИХ АДРЕС ПРИ КЕШНУВАННІ

Приклад у цьому розділі припускає, що кеш-пам'ять використовує адреси фізичної пам'яті для позначки записів кеша. Це означає, що в системах із підтримкою віртуальної пам'яті зі сторінковою організацією адреси, які використовуються під час пошуку в кеші, є результатом трансляції віртуальних адрес у фізичні. Розробник процесора повинен вибрати, які адреси слід використовувати для цілей кешування – віртуальні або фізичні.



У кеш-пам'яті сучасних процесорів на одному або декількох рівнях кешування нерідко використовуються теги віртуальних адрес, а на інших рівнях – теги фізичних адрес. Однією з переваг використання тегів віртуальних адрес є швидкість, оскільки під час звернення до кешу не потрібна трансляція віртуальної адреси у фізичну. Як ми бачили, під час трансляції віртуальної адреси у фізичну виконується пошук у буфері TLB і, можливо, пошук за таблицею сторінок у разі промаху TLB. Однак використання тегів віртуальних адрес створює інші проблеми, такі як **множинність посилань**, що виникає, коли одна і та ж віртуальна адреса посилається на різні фізичні адреси.

Як і в багатьох інших аспектах оптимізації продуктивності процесора, це компроміс, на який слід зважати під час проектування системи кешування.

Цей приклад було спрощено за рахунок припущення, що виконання інструкцій відбувається лінійно від початку і до кінця циклу без будь-яких обхідних шляхів. У реальному коді часто відбуваються виклики функцій у різних областях пам'яті програми та звернення до системних бібліотек.

На додачу до цих факторів інформація в кеші часто перезаписується під час виконання інших системних дій, таких як обробка переривань і перемикання контексту потоку, що призводить до більш високої частоти промахів кеша. Це впливає на продуктивність програми, оскільки код основної гілки повинен виконувати додаткові звернення до DRAM для перезавантаження кеша після кожного відхилення, яке призводило до змін записів у кеші.

Одним зі способів зменшити наслідки відхилень від прямолінійного виконання коду є налаштування декількох кешів, що працюють паралельно. Таку конфігурацію, яка обговорюється в наступному розділі, називають набірно-асоціативним **кешем**.

Набірно-асоціативний кеш

У двоканальному набірно-асоціативному кеші пам’ять ділиться на два кеші однакового розміру. Кожен з них містить половину загальної кількості записів кеша з прямим відображенням того ж загального розміру. При кожному доступі до пам'яті апаратні засоби звертаються до обох кешів паралельно, і потрапляння може статися в будь-якому з них. Наступна схема ілюструє одночасне порівняння тегу 32-розрядної адреси з тегами, що містяться у двох I-кешах L1.

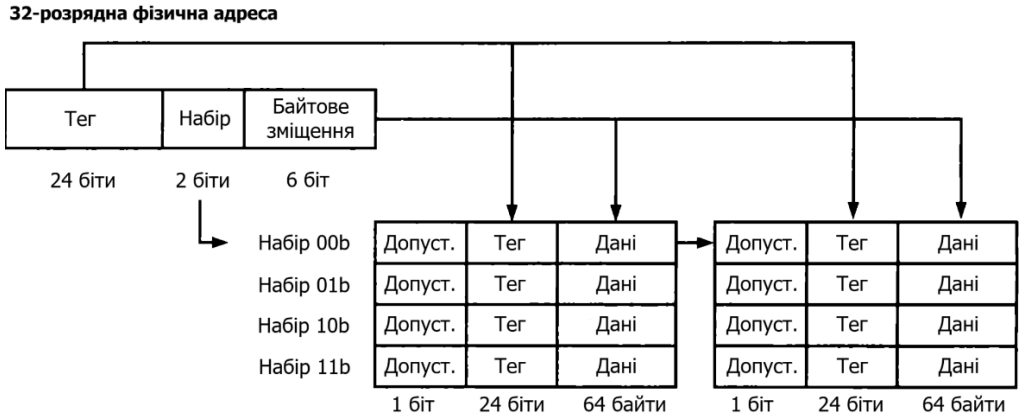


Рис 8.4. *Робота набірно-асоціативного кеша*

Конфігурація кеша, показана тут, має той самий розмір рядка кеша (64 байти), що й кеш на рис. 8.3, але вдвічі менше наборів на кеш. Загальний розмір кеш-пам'яті такий самий, як і в попередньому прикладі: 64 байти на рядок, помножені на 4 рядки, помножені на 2 кеші, дорівнюють 512 байтам. Оскільки тепер існує чотири набори, поле "Набір" у фізичній адресі зменшується до 2 біт, а поле "Тег" збільшується до 24 біт. Кожен набір складається з двох рядків кеша, по одному в кожному з двох кешів.

При кеш-промаху логіка керування пам'яттю повинна вибрати, яку з двох таблиць кеша використовувати як місце призначення для даних, що завантажуються з DRAM.

Загальноприйнятий метод полягає в тому, щоб відстежувати, до якого з відповідних наборів у двох таблицях довше не було звернень, і перезаписати цей запис. Ця політика заміщення, звана **LRU** (least recently used – заміна найдавнішого використовуваного елемента), вимагає апаратної підтримки для відстеження того, який рядок кешу не був затребуваний довше всього. Політика LRU заснована на евристиці тимчасової локальності, яка свідчить, що дані, до яких не зверталися протягом тривалого часу, з меншою ймовірністю будуть затребувані знову найближчим часом.

Інший спосіб вибору між двома таблицями – просто чергувати їх під час послідовних вставках у кеш.

Апаратні засоби для реалізації цієї політики заміни простіші, ніж для політики LRU, але такий підхід може вплинути на продуктивність через те, що довільний вибір рядка може бути невірним. Вибір політики заміщення вмісту кеша являє собою ще одну область компромісу між підвищеною складністю апаратних засобів і поступовим підвищенням продуктивності.

У двоканальному набірно-асоціативному кеші два рядки кеша з різних фізичних місць розташування зі збіжними полями "Набір" можуть бути присутніми в кеші одночасно, за умови, що обидва рядки кеша допустимі. Водночас, порівняно з кешем із прямим відображенням того ж загального розміру, кожен із двоканальних набірно-асоціативних кешів має вдвічі менший розмір. Це ще один компроміс проєктування: кількість унікальних значень поля "Набір", що можуть зберігатися у двоканальному набірно-асоціативному кеші, менша порівняно з прямим відображенням, але водночас можливе одночасне кешування кількох рядків кеша з однаковими полями "Набір".

Конфігурація кеша, що забезпечує найкращу загальну продуктивність системи, залежатиме від особливостей доступу до пам'яті, пов'язаних із типами операцій оброблення, які виконуються в системі.

Набірно-асоціативний кеш може містити більше двох кешів, розглянутих у цьому прикладі. Сучасні процесори часто мають 4, 8 або 16 паралельних кешів, які називають 4-канальними, 8-канальними та 16-канальними набірно-асоціативними кешами. Ці кеші називаються набірно-асоціативними, тому що поле "Тег" адреси одночасно відноситься до всіх рядків кеша в наборі. Кеш із прямим відображенням реалізує одноканальний набірно-асоціативний кеш.

Перевага багатоканальних набірно-асоціативних кешів порівняно з кешами з прямим відображенням полягає в тому, що вони, як правило, мають вищу частоту кеш-попадань, що в більшості практичних випадків веде до підвищення продуктивності системи порівняно з кешами з прямим відображенням. Якщо багатоканальні набірно-асоціативні кеші забезпечують кращу продуктивність, ніж одноканальні кеші з прямим відображенням, то чому б тоді не підвищити рівень асоціативності? Доведена до межі, ця прогресія закінчується повністю асоціативним кешуванням.

Повністю асоціативний кеш

Припускаючи, що кількість рядків у кеші дорівнює ступеню двійки, повторюваний поділ загальної кеш-пам'яті на більшу кількість менших паралельних кешів, доки кожний кеш не міститиме лише один рядок, призводить до отримання **повністю асоціативного кешу**.

У нашому прикладі 512-байтового кеша з 64 байтами на рядок кеша цей процес призведе до створення восьми паралельних кешів, кожен з яких містить тільки один набір.

У цій архітектурі кожне звернення до пам'яті призводить до паралельного порівняння зі значеннями тегів, що зберігаються у всіх рядках кеша. Повністю асоціативний кеш з використанням ефективної політики заміни, такої як LRU, може забезпечити дуже високу частоту кеш-попадань, хоча й за значних витрат на складність схеми та відповідне споживання площі кристалу.

У рішеннях, чутливих до енергоспоживання, таких як мобільні пристрої з живленням від акумулятора, підвищення складності схеми повністю асоційованого кеша призводить до збільшення розряду акумулятора. У настільних комп'ютерах і хмарних серверах енергоспоживання процесора повинно бути зведене до мінімуму, щоб уникнути необхідності в надзвичайних заходах для охолодження і звести до мінімуму рахунки за електроенергію для постачальників хмарних послуг, які експлуатують тисячі серверів. Зважаючи на такі витрати, повністю асоціативний кеш рідко використовується як кеш інструкцій і даних у сучасних процесорах.

Концепція, що лежить в основі повністю асоціативного кеша, може здатися знайомою, оскільки це та сама концепція, яка застосовується в буфері TLB, описаному в *розділі 7*. TLB зазвичай являє собою повністю асоціативний кеш, що містить результати трансляції віртуальних адрес у фізичні. Використання повністю асоціативного кешування в TLB має свої недоліки, описані в попередньому абзаці: складність схеми, потреба в додатковій площі кристала та підвищене енергоспоживання, – однак перевага у вигляді підвищення продуктивності, що забезпечує TLB, є настільки істотною, що повну асоціативність застосовують практично у всіх високопродуктивних процесорах, де реалізована віртуальна пам'ять зі сторінковою організацією.

Наше обговорення до цього моменту було зосереджене на кешуванні інструкцій, яке зазвичай є процесом із доступом тільки для читання. Функціональні вимоги до кешу даних багато в чому аналогічні вимогам до кешу інструкцій з одним важливим доповненням: крім читання з пам'яті даних процесор повинен бути вільний для запису в пам'ять даних. Це тема наступного розділу.

Політики запису в кеш процесора

У процесорах із розділеною кеш-пам'яттю D-кеш L1 аналогічний за структурою I-кешу L1, за винятком того, що схема повинна дозволяти процесору не тільки зчитувати дані з пам'яті, а й записувати їх у пам'ять. Кожного разу, коли процесор записує значення даних у пам'ять, він повинен оновити рядок D-кеша L1, який містить цей елемент даних, і в якийсь момент часу він також повинен оновити осередок DRAM у фізичній пам'яті, який містить цей рядок кеша. Запис у пам’ять DRAM, як і читання з неї, є повільним процесом порівняно зі швидкістю запису в D-кеш L1.

Нижче перелічено найпоширеніші політики запису в кеш у сучасних процесорах.

- **Наскрізний запис (write-through).** Ця політика передбачає негайне оновлення DRAM щоразу, коли процесор записує дані в пам'ять.
- **Відкладений запис (write-back).** У разі використання цієї політики змінені дані утримуються в кеші доти, доки відповідний рядок не буде витіснено з кеша. Для кожного рядка кеша з відкладеним записом має бути передбачений додатковий біт стану, що вказує, чи були змінені дані з моменту їхнього зчитування з DRAM. Цей біт називається **бітом факту запису**. Встановлений стан цього біта вказує, що дані в цьому рядку були змінені, і система повинна записати їх у DRAM, перш ніж відповідний рядок кеша може бути звільнений для повторного використання.

Політика відкладеного запису зазвичай призводить до підвищення продуктивності системи, оскільки дає процесору можливість виконувати кілька операцій запису в один і той самий рядок кеша без необхідності звертатися до основної пам'яті під час кожного запису. У системах із кількома ядрами або кількома процесорами використання кешування з відкладеним записом створює складнощі, оскільки D-кеш L1, який належить ядру, яке виконує запис у комірку пам'яті, міститиме дані, що відрізняються від вмісту відповідної комірки DRAM та від будь-яких копій у кешах інших процесорів або ядер. Така невідповідність вмісту пам'яті зберігатиметься доти, доки цей рядок кеша не буде записано в DRAM і оновлено в будь-якому з кешів процесора, де він знаходиться.

Як правило, неприпустимо, щоб різні ядра процесора витягували різні значення під час оброблення однієї й тієї самої комірки пам'яті, тому необхідно передбачити рішення, яке гарантуватиме, що всі процесори завжди бачать одні й ті самі дані в кожній комірці пам'яті. Завдання підтримання ідентичних уявлень вмісту пам'яті на кількох процесорах називається завданням узгодження кеша.

У багатоядерному процесорі можна було б розв'язати цю проблему шляхом організації спільного доступу всіх ядер до одного й того самого D-кешу L1, але сучасні багатоядерні процесори зазвичай реалізують окремий D-кеш L1 для кожного ядра, щоб довести швидкість доступу до максимуму. У багатопроцесорних системах зі

спільною основною пам'яттю, де процесори розташовані на окремих інтегрованих схемах, і в багатоядерних процесорах, які не використовують спільний кеш L1, це завдання відрізняється більшою складністю.

У деяких багатопроцесорних архітектурах для підтримання узгодженості кеша застосовується відстеження. Спостерігаючи за пам’яттю, процесор відстежує операції запису в пам'ять, що виконуються іншими процесорами. Коли відбувається запис у комірку пам'яті, яка присутня в кеші відстежуючого процесора, відстежуючий пристрій виконує одну з двох дій: він може зробити недійсним свій рядок кеша, встановивши біт допустимості в стан false, або занести в цей рядок дані, записані іншим процесором. Якщо рядок кеша визнано неприпустимим, наступне звернення до діапазону адрес цього рядка викличе доступ до DRAM для отримання даних, змінених іншим процесором.

Відстеження може бути ефективним у системах з обмеженим числом процесорів, але його складно масштабувати для систем, що містять десятки або сотні процесорів. Це пов’язано з тим, що кожен процесор повинен постійно відстежувати операції запису всіх інших процесорів. У системах з великою кількістю процесорів мають бути реалізовані інші, складніші протоколи узгодження кеша.

Кеші процесора другого і третього рівнів

До теперішнього моменту наше обговорення було зосереджено на кешах інструкцій і даних першого рівня – L1. Ці кеші спроектовані так, щоб бути максимально швидкими, але акцент на швидкості обмежує їхній розмір через складність схеми кеша і вимог до живлення.

Через велику різницю в затримці кеша L1 і DRAM розумно замислитися над питанням: чи може проміжний рівень кешування між L1 і DRAM підвищити продуктивність порівняно з процесором, який містить тільки кеш L1? Відповідь – так, додавання кеша L2 забезпечує істотне підвищення продуктивності.

На кристалах сучасних високопродуктивних процесорів зазвичай міститься банк кеша L2 значного обсягу. На відміну від кеша L1, кеш L2 зазвичай об'єднує інструкції та дані в одній ділянці пам'яті, представляючи архітектуру фон Неймана, а не характерний для гарвардської архітектури розподіл у структурі кеша L1.

Кеш L2 зазвичай працює повільніше, ніж кеш L1, але все ж набагато швидше, ніж прямий доступ до DRAM. Кеш L2 використовує бітові комірки SRAM з тією самою базовою схемою, що показана на рис. 8.1, однак у схемному рішенні L2 приділяється особлива увага зменшенню площі, що займається на кристалі, із розрахунку на біт, і зниженню енергоспоживання порівняно зі схемним рішенням L1. Ці модифікації дозволяють зробити кеш L2 набагато більшим за обсягом, ніж кеш L1, але вони також роблять його значно повільнішим.

Типовий кеш L2 може бути в чотири або вісім разів більшим, ніж I-кеш і D-кеш L1 разом узяті, водночас час доступу в нього буде в 2-3 рази більшим, ніж у L1. Залежно від конструкції процесора кеш L2 може включати або не включати в себе кеш L1. **Інклюзивний кеш L2**, крім інших рядків, завжди містить рядки кеша, наявні в кеші L1.

Як ми вже бачили, щоразу, коли процесор звертається до пам'яті, він спочатку звертається до кешу L1. Якщо відбувається промах кешу L1, наступним пунктом для перевірки є кеш L2. Оскільки L2 більший за L1, існує значна ймовірність того, що дані будуть знайдені саме там. Якщо шуканий рядок кеша відсутній у L2, потрібен доступ до DRAM. Процесори зазвичай виконують деякі з цих кроків паралельно, щоб гарантувати, що промах кеша не призведе до тривалого послідовного пошуку запитаних даних.

Кожен промах кеша L1 і L2 призводить до виконання операції читання DRAM, яка заповнює відповідні рядки кеша в L1 і L2 за умови, що L2 є інклюзивним. За кожною виконуваною процесором операцією запису в пам'ять даних рано чи пізно має відбутися оновлення як L1, так і L2. D-кеш L1 реалізує політику запису в кеш (зазвичай це політика наскрізного або відкладеного запису), щоб визначити, коли має бути виконано оновлення кеша L2. У кеші L2, аналогічним чином, реалізовано власну політику запису, що допомагає визначити, коли слід записувати змінені рядки кеша в DRAM.

Якщо використання двох рівнів кеша допомагає підвищити продуктивність, чи варто обмежуватися лише ними? Насправді в більшості сучасних високопродуктивних процесорів реалізовано три (або навіть більше!) рівні кеша на кристалі. Як і під час переходу від L1 до L2, перехід від L2 до L3 призводить до збільшення обсягу пам'яті та зменшення швидкості доступу. Як і L2, кеш L3 зазвичай об'єднує інструкції та дані в одній області пам'яті. У процесорах споживчого рівня кеш L3 зазвичай являє собою SRAM об'ємом у кілька мегабайт із часом доступу в 3-4 рази повільнішим, ніж кеш L2.

ПОГЛЯД ПРОГРАМІСТА НА КЕШ-ПАМ’ЯТЬ



Розробникам програмного забезпечення не потрібно вживати якихось кроків, щоб скористатися перевагами кеш-пам'яті, однак розуміння середовища, у якому працює їхнє програмне забезпечення, може допомогти підвищити продуктивність коду, який виконують процесори з кількома рівнями кеш-пам'яті. За наявності достатньої гнучкості визначення розмірів структур даних і структурування внутрішніх циклів коду для роботи в межах передбачуваних розмірів кешів L1, L2 і L3 може призвести до значного збільшення швидкості виконання.

На продуктивність будь-якого фрагмента коду впливає безліч чинників, пов'язаних з архітектурою процесора та поведінкою системного програмного забезпечення, тому найкращий спосіб визначити оптимальний алгоритм серед кількох життєздатних альтернатив – це ретельне порівняння продуктивності кожного з них.

Багаторівнева ієрархія кеш-пам’яті в сучасних процесорах призводить до значного підвищення продуктивності порівняно з аналогічною системою без кеш-пам'яті. Кешування дозволяє найшвидшим процесорам більшу частину часу працювати з мінімальною затримкою доступу до DRAM. Незважаючи на те що кеш-пам'ять значно ускладнює конструкцію процесора і споживає багато цінного місця на кристалі та енергії, виробники процесорів вирішили, що використання багаторівневої архітектури кеш-пам'яті цілком виправдовує ці витрати.

Кеш-пам’ять прискорює виконання інструкцій за рахунок зменшення затримки доступу до пам’яті при вилученні інструкцій і даних, на які посилаються ці інструкції.

Наступна область підвищення продуктивності, яку ми розглянемо, – це можливості оптимізації в процесорі, що допомагають збільшити швидкість виконання інструкцій. Основним методом, який застосовується в сучасних процесорах для досягнення приросту продуктивності таким способом, є конвеєризація.

Конвеєрна обробка інструкцій

Перш ніж представити конвеєрну обробку, давайте розіб'ємо виконання однієї інструкції процесора на послідовність окремих кроків.

- **Витяг.** Пристрій керування процесора отримує доступ до адреси пам'яті, за якою розташована наступна інструкція для виконання, як визначено попередньою інструкцією, або із зумовленого значення внаслідок скидання програмного лічильника одразу після увімкнення живлення, або у відповідь на переривання. Зчитавши цю адресу, пристрій керування завантажує опкод інструкції у внутрішній регістр інструкцій процесора.
- **Декодування.** За отриманим опкодом пристрій керування визначає дії, які необхідно зробити під час виконання інструкції. Ці дії можуть включати в себе звернення до АЛУ і вимагати доступу до регістрів або комірок пам'яті для читання або запису.
- **Виконання.** Пристрій керування виконує запитану операцію, викликаючи за необхідності операцію АЛУ.
- **Запис результатів.** Пристрій керування записує результати виконання інструкції в регістри або комірки пам'яті. Після цього в програмний лічильник завантажується адреса наступної інструкції, яка має бути виконана.

Процесор виконує повторюваний цикл вилучення, декодування, виконання і запису з моменту подачі живлення до вимкнення комп'ютера. У відносно простому процесорі, такому як 6502, кожен із цих етапів виконується процесором як фактично ізольовані послідовні операції.

З точки зору програміста, кожна інструкція виконає всі ці етапи перед початком виконання наступної інструкції. Після завершення запису результати та побічні ефекти кожної інструкції доступні в регістрах, пам’яті та прапорах процесора для негайного використання наступною інструкцією. Це проста модель виконання, і програми, що працюють, можуть з упевненістю вважати, що коли починається виконання інструкції, всі дії попередніх інструкцій завершені.

Схема на рис. 8.5 є прикладом виконання процесором інструкції, для кожного з етапів опрацювання якої (вилучення – декодування – виконання – запис) потрібен один тактовий цикл. Зверніть увагу, що кожен крок на цій діаграмі позначено першою літерою назви етапу: І, Д, В або З.



Рис 8.5. *Послідовне виконання інструкцій*

Для виконання кожної інструкції потрібно чотири такти. На апаратному рівні процесор, представлений на рис. 8.5, складається з чотирьох підсистем виконання, кожна з яких стає активною під час одного з чотирьох тактових циклів інструкції. Логіка опрацювання, пов’язана з кожним із цих етапів, передбачає зчитування вхідної інформації з пам'яті та з регістрів процесору і збереження проміжних результатів у засувках для використання на наступних етапах виконання. Після завершення виконання кожної інструкції наступна інструкція починає виконуватися в наступному тактовому циклі.

Кількість виконаних інструкцій за такт процесора (instructions per clock, IPC) являє собою показник продуктивності, що характеризує, наскільки швидко процесор виконує інструкції відносно своєї тактової частоти. Процесору в прикладі на рис. 8.5 потрібно чотири такти на одну інструкцію, тобто показник IPC для нього дорівнює 0,25.

Можливість підвищення продуктивності в цьому прикладі виникає через те, що схеми, задіяні в кожному з чотирьох етапів, простоюють, поки виконуються інші три етапи. Припустимо, що замість цього, щойно схема вилучення закінчує вилучення

однієї інструкції, вона негайно починає вилучення наступної інструкції. На рис. 8.6 показано процес, коли апаратні засоби, що беруть участь у кожному з чотирьох етапів виконання інструкції, переходять від обробки однієї інструкції до наступної на кожному тактовому циклі.

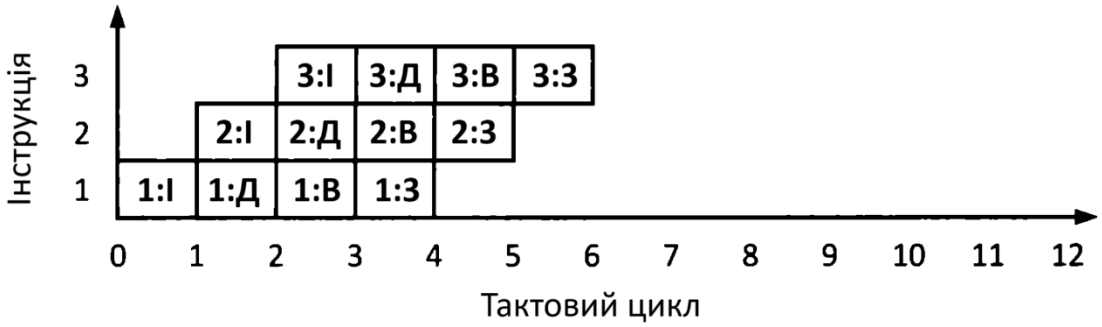


Рис 8.6. *Конвеєрне виконання інструкцій*

Ця процедура виконання називається **конвеєрною**, оскільки інструкції надходять і проходять етапи виконання від початку до кінця, подібно виробам, які переміщуються по реальному конвеєру. Конвеєр процесора містить безліч інструкцій, що одночасно перебувають на різних етапах виконання. Причина, що спонукала піти на це, очевидна з попереднього прикладу: процесор тепер виконує одну інструкцію за тактовий цикл, а IPC становить 1,0, що дає чотири-кратне збільшення швидкості порівняно з моделлю виконання без конвеєрного опрацювання, представленою на рис. 8.5. Аналогічні рівні підвищення продуктивності досягаються при обробленні реальних даних за допомогою методів конвеєризації.

Крім накладення виконання послідовних інструкцій за допомогою конвеєризації, можуть існувати й інші можливості продуктивного використання підсистем процесора, які в іншому разі можуть простоювати. Інструкції процесора діляться на кілька різних категорій, кожна з яких вимагає задіяння різних частин схеми процесора. Ось деякі приклади.

- **Інструкції розгалуження.** Інструкції умовного і безумовного розгалуження керують програмним лічильником, щоб встановити адресу наступної інструкції, яка підлягає виконанню.
- **Інструкції цілочисельних обчислень.** Ці інструкції, що реалізують цілочисельну арифметику і побітові операції, звертаються до цілочисельної частини АЛУ.
- **Інструкції обчислень із плаваючою комою.** Операції з плаваючою комою в процесорах, які забезпечують апаратну підтримку цих операцій, зазвичай використовують схему, відокремлену від цілочисельної частини АЛУ.

За рахунок підвищення складності логіки планування інструкцій процесора може виявитися можливим ініціювати виконання двох інструкцій за один і той самий тактовий цикл, якщо вони використовують незалежні ресурси обробки. Наприклад,

звичайною практикою є відправлення процесорами інструкцій обчислень із плаваючою комою в **математичний співпроцесор** (floating point unit, FPU) для виконання паралельно з інструкціями без плаваючої коми, що обробляються ядром основного процесора. На практиці багато сучасних процесорів містять кілька копій підсистем, таких як цілочисельні АЛУ, для підтримки одночасного виконання кількох інструкцій. У цій архітектурній парадигмі, яка називається **обробкою з множинною видачею інструкцій** (multiple-issue processing), процесор ініціює виконання кількох інструкцій одночасно. На рис. 8.7 показано приклад множинної видачі, за якої на кожному такті процесора ініціюється виконання двох інструкцій.

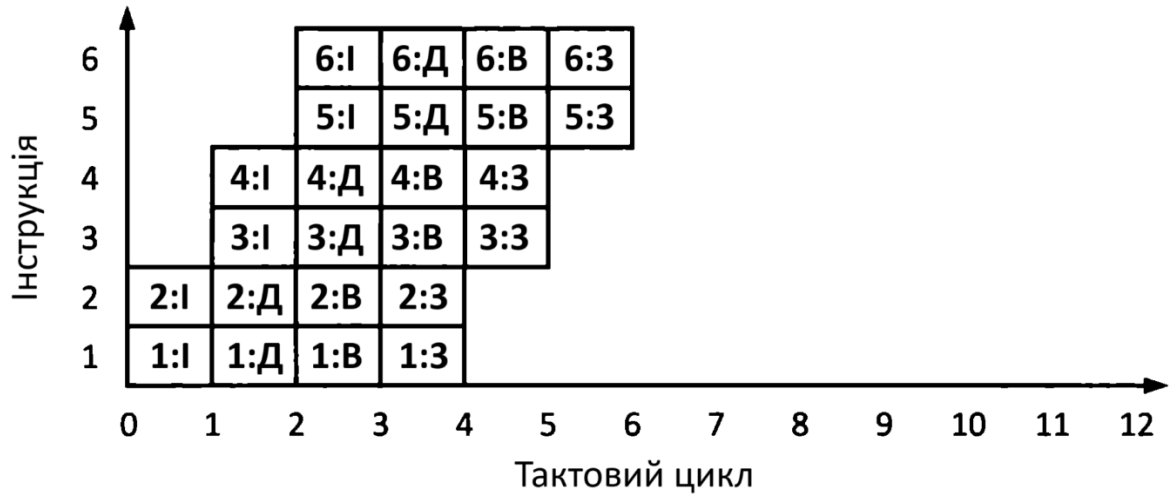


Рис 8.7. *Конвеєрне виконання з множинною видачею інструкцій*

Ця модель виконання подвоює кількість інструкцій, що виконуються за такт на одноканальному конвеєрі рис. 8.6, що дає змогу довести IPC до 2,0. Цей підхід представляє **суперскалярний процесор**, який може видавати (інакше кажучи, починати виконання) більш ніж одну інструкцію за такт. **Скалярний процесор**, для порівняння, може видавати тільки одну інструкцію за такт. Для того щоб прояснити ситуацію: на рис. 8.5 і 8.6 представлено поведінку скалярного процесора, а на рис. 8.7 – суперскалярне опрацювання. Суперскалярний процесор реалізує **паралелізм на рівні інструкцій** (instruction-level parallelism, ILP) для збільшення швидкості виконання. Практично всі сучасні високопродуктивні процесори загального призначення є суперскалярними.

Суперконвеєризація

На прикладі розглянутої раніше скалярної, не конвеєрної моделі обробки, представленої на рис. 8.5, ми можемо обговорити питання вибору тактової частоти процесора. За відсутності побоювань з приводу енергоспоживання і розсіювання тепла бажано, як правило, використовувати максимально можливу тактову частоту процесора. Верхня межа тактової частоти для процесора на рис. 8.5 визначається найтривалішим за часом шляхом через кожну з чотирьох підсистем, що беруть участь в етапах виконання інструкцій. Різні інструкції можуть мати абсолютно різні вимоги

до часу виконання. Наприклад, для виконання інструкції зняття прапора процесора потрібен зовсім невеликий час, у той час як для одержання вихідних даних інструкції 32-розрядного ділення може знадобитися набагато більше часу.

Неефективно обмежувати загальну швидкість виконання інструкцій системи на основі найгіршого часу виконання однієї інструкції. Замість цього розробники процесорів шукають можливості розбити виконання складних інструкцій на деяку кількість послідовних кроків. Цей підхід називається **суперконвеєризацією**. Суперконвеєризація полягає у збільшенні кількості етапів конвеєра за рахунок розбивки складних етапів на безліч більш простих. Суперконвеєризація – це, по суті, конвеєр процесора з безліччю етапів, що можливо обчислюються десятками. На додаток до того, що сучасні високопродуктивні процесори є суперскалярними, вони, як правило, підтримують і суперконвейєризацію.

Розбиття конвеєра на більшу кількість етапів суперконвеєра дає змогу спростити кожен етап, скорочуючи час, необхідний для виконання кожного етапу. Завдяки швидше виконуваним етапам можна збільшити тактову частоту процесора. Суперконвеєризація забезпечує збільшення швидкості виконання інструкцій відповідно до відсоткового збільшення тактової частоти процесора.

Набори інструкцій процесорів для **комп'ютерів зі скороченим набором інструкцій** (reduced instruction set computer, RISC) розроблено для підтримки ефективної конвеєризації. Більшість RISC-інструкцій виконують прості операції, як-от переміщення даних між регістрами та пам'яттю або додавання вмісту двох регістрів. RISC-процесори зазвичай мають коротші конвеєри порівняно з процесорами для **комп'ютерів із повним набором інструкцій** (complex instruction set computer, CISC). CISC-процесори та їхні ширші та складніші набори інструкцій отримують вигоду від довших конвеєрів, розбиваючи довготривалі інструкції, що виконуються, на серію послідовних етапів.

Здебільшого складність реалізації ефективного конвеєрного опрацювання в процесорах, які використовують застарілі набори інструкцій, таких як x86, полягає в тому, що первісний набір інструкцій не повністю враховував потенціал більш пізніх удосконалень, таких як суперскалярне опрацювання і суперконвеєризація. У результаті в сучасних 386-сумісних процесорах значна частина площі кристала виділена для складної логіки, що реалізує ці функції підвищення продуктивності для інструкцій, які не були призначені для роботи в такому середовищі.

Якщо розбиття конвеєра на один або два десятки етапів призводить до істотного підвищення продуктивності, чому б не продовжити це розбиття на сотні або навіть тисячі дрібніших етапів, щоб досягти ще більшої продуктивності? Відповідь: наявність конфліктів конвеєризації.

Конфлікти конвеєризації

Реалізація конвеєрного оброблення в процесорі загального призначення не така проста, як може здатися з попереднього обговорення цієї теми. Якщо результат інструкції залежить від результату попередньої інструкції, то поточній інструкції доводиться чекати, поки не стане доступний результат попередньої інструкції. Розглянемо наступний код x86:

```
inc eax
add ebx, eax
```

Припустимо, що ці дві інструкції виконуються на процесорі, що працює як показано на рис. 8.6 (одноканальна конвеєризація). Перша інструкція збільшує вміст регістра `eax` на одиницю, а друга - додає отримане значення до вмісту регістра `ebx`. Інструкція `add` не може виконати операцію додавання доти, доки не буде доступний результат операції збільшення на 1 у попередній інструкції. Якщо етап виконання другої інструкції (позначений як **2:B** на рис. 8.6) не може бути виконаний доти, доки не завершиться етап запису результату першої інструкції (**1:3** на рис. 8.6), в етапу **2:B** немає іншого вибору, окрім як дочекатися завершення етапу **1:3**. Ситуації, подібні до цієї, коли конвеєр не може обробити наступний етап виконання інструкції через відсутність необхідної інформації, називають **конфліктами конвеєризації**.

Одним зі способів, за допомогою якого конвеєри процесорів вирішують цю проблему, є використання обхідних шляхів. Після завершення етапу виконання першої інструкції (позначеного як **1:B** на рис. 8.6), збільшене на 1 значення в регістрі `eax` було обчислено, але ще не було записано в регістр `eax`. Етапу виконання другої інструкції (**2:B** на рис. 8.6) як вхідні дані потрібне збільшене на 1 значення регістра `ean`. Якщо логіка конвеєра робить результат етапу **1:B** доступним для етапу **2:B** без попереднього запису в регістр `eax`, то виконання другої інструкції можна завершити без затримки. Використання такого типу спрощення для передачі даних між вихідною та цільовою інструкціями без очікування завершення вихідної інструкції називається обходом. Обходи широко використовуються в архітектурах сучасних процесорів, щоб забезпечити максимально ефективну роботу конвеєра.

У деяких ситуаціях обхід неможливий, оскільки потрібний результат просто не може бути обчислений до того, як цільова інструкція буде готова його використовувати. У цьому разі необхідно призупинити виконання цільової інструкції і почекати доставки результату вихідної інструкції. Це призводить до того, що етап виконання конвеєра простоює, що зазвичай призводить до поширення періоду простою на інші етапи конвеєра.

Ця затримка, що поширюється поетапно, називається **конвеєрною бульбашкою** за аналогією з повітряною бульбашкою, що проходить по трубопроводу з рідиною. Наявність бульбашок у послідовності обробки знижує ефективність конвеєризації.

Бульбашки погано впливають на продуктивність, тому розробники конвеєрних процесорів докладають значних зусиль, щоб по можливості їх уникнути. Одним із способів зробити це є **позачергове виконання інструкцій**, що позначається **OoO** (Out-of-Order instruction execution). Розглянемо дві інструкції, згадані в попередньому прикладі, але тепер за ними слідує третя інструкція:

```
inc eax
add ebx, eax
mov ecx, edx
```

Третя інструкція не залежить від результатів виконання попередніх інструкцій. Замість використання обходу, щоб уникнути затримки, процесор може вдатися до позачергового виконання інструкцій для виключення конвеєрної бульбашки. Підсумкова послідовність виконання інструкцій матиме такий вигляд:

```
inc eax
mov ecx, edx
add ebx, eax
```

Результат виконання цих трьох інструкцій один і той самий, але різниця в часі між першою і третьою інструкціями збільшилася, що знімає ймовірність виникнення конвеєрного міхура, а якщо він все ж виникне, то такий порядок, принаймні, скорочує його тривалість.

Метод позачергового виконання інструкцій передбачає виявлення залежностей між інструкціями і перебудовує порядок їх виконання таким чином, щоб отримати ті самі загальні результати, але не в тому порядку, у якому вони були спочатку закодовані. Деякі процесори (зазвичай CISC) виконують це впорядкування інструкцій у реальному часі під час виконання програми. Інші архітектури (зазвичай RISC) покладаються на інтелектуальні компілятори мов програмування, які виконують перестановку інструкцій у процесі складання програмного забезпечення, щоб звести до мінімуму утворення конвеєрних бульбашок. Перший підхід вимагає значних вкладень у логіку оброблення та пов'язану з нею корисну площу кристала для виконання перевпорядкування "на льоту", тоді як другий підхід спрощує логіку процесора, але істотно ускладнює роботу програмістів на асемблері, які в цьому випадку несуть відповідальність за те, щоб конвеєрні бульбашки не надто погіршували продуктивність виконання програми.

У деяких CISC-процесорах для реалізації переваг щодо продуктивності, властивих RISC-архітектурам, використовуються мікрооперації та перейменування регістрів. Ці методи представлено в наступному розділі.

Мікрооперації та перейменування регістрів

Архітектура набору інструкцій x86 представляє особливу проблему для розробників процесорів. Незважаючи на те що історія архітектури x86 налічує вже кілька десятиліть, вона залишається основою персональних і комерційних обчислень. Оскільки конфігурація CISC містить усього вісім регістрів (у 32-розрядній версії x86), методи високоефективної конвеєризації інструкцій і використання великої кількості регістрів, доступних в архітектурах RISC, набагато менш корисні в оригінальній архітектурі x86.

Для того щоб скористатися деякими перевагами методики RISC, розробники процесорів x86 вирішили представити інструкції x86 у вигляді послідовностей мікрооперацій. Мікрооперації, скорочено **μop** (вимовляється як "*майкро-оп*"), є під етапом інструкції процесора. Прості інструкції x86 поділяються на 1-3 мікрооперації, а більш складні – не більшу за їх кількість. Поділ інструкцій на мікрооперації забезпечує вищий рівень деталізації для оцінки залежностей від результатів інших мікрооперацій і підтримує збільшення паралелізму виконання.

Разом із підтримкою мікрооперацій сучасні процесори зазвичай надають додаткові внутрішні регістри (числом у десятки або навіть сотні) для зберігання проміжних результатів мікрооперацій. Ці регістри містять частково обчислені результати, призначені для перенесення у фізичні регістри процесора після завершення всіх мікрооперацій інструкції. Використання цих внутрішніх регістрів називається **перейменуванням регістрів**. Кожен перейменований регістр має відповідне значення тегу, що вказує, який фізичний регістр процесора він у підсумку займе. За рахунок збільшення кількості перейменованих регістрів, доступних для проміжного зберігання, збільшуються можливості паралелізму інструкцій.

Декілька мікрооперацій можуть перебувати на різних стадіях виконання в будь-який момент часу. Залежності інструкції від результату попередньої інструкції блокують виконання мікрооперації доти, доки не будуть доступні її вхідні дані, які зазвичай передаються за допомогою механізму обходу, описаного в попередньому розділі. Як тільки всі необхідні вхідні дані будуть доступними, мікрооперація буде запланована для виконання.

Цей режим роботи являє собою модель **обробки за потоком даних**. Таке опрацювання дає змогу виконувати паралельні операції в суперскалярній архітектурі, ініціюючи виконання мікрооперацій після розв’язання будь-яких залежностей за даними.

Високопродуктивні процесори здійснюють декодування інструкцій у мікрооперації після вилучення кожної інструкції. У деяких мікросхемах результати цього декодування зберігаються в кеші інструкцій рівня 0, розташованому між процесором і I-кешем L1. I-кеш L0 забезпечує максимально швидкий доступ до попередньо декодованих мікрооперацій для виконання внутрішніх циклів коду з максимально можливою швидкістю. Завдяки кешуванню декодованих мікрооперацій процесор уникає необхідності повторювати етапи конвеєра, що виконують декодування інструкцій, для кожного наступного доступу до однієї й тієї самої інструкції.

На додаток до конфліктів, пов'язаних із залежностями за даними між інструкціями, другим ключовим джерелом конфліктів конвеєризації є виникнення умовного розгалуження, обговорюваного в наступному розділі.

Умовне розгалуження

Умовне розгалуження вносить істотні труднощі в процес конвеєрного оброблення. Адресу інструкції, наступної за інструкцією умовного розгалуження, не можна підтвердити доти, доки не буде оцінено умову розгалуження.

Є дві можливості: якщо умова розгалуження не виконується, процесор виконає інструкцію, яка слідує за інструкцією умовного розгалуження. Якщо умова розгалуження виконується, наступна інструкція знаходиться за адресою, вказаною в інструкції.

Для розв'язання проблем, пов'язаних з умовним розгалуженням у конвеєрних процесорах, використовується кілька методів. Ось деякі з них.

- Коли це можливо, слід взагалі уникати розгалуження. Розробники програмного забезпечення можуть розробляти алгоритми з внутрішніми циклами, які зводять до мінімуму або зовсім усувають умовний код. Оптимізуючи компілятори прагнуть перебудувати і спростити послідовність операцій, щоб зменшити негативні наслідки умовного розгалуження.
- Процесор може відкласти витяг наступної інструкції доти, доки не буде оцінено умову розгалуження. Зазвичай це призводить до утворення бульбашок у конвеєрі, знижуючи продуктивність.
- Процесор може виконати обчислення умови розгалуження на якомога більш ранніх етапах конвеєра. Завдяки цьому можна швидше ідентифікувати правильну гілку, забезпечуючи виконання з меншою затримкою.
- Деякі процесори намагаються вгадати результат умови розгалуження і розпочинають попереднє виконання інструкцій за цим вгаданим шляхом. Цей метод називається **прогнозуванням розгалуження**. Якщо припущення виявляється хибним, процесор має видалити результати неправильно виконаних інструкцій із конвеєра (ця операція називається **очищенням конвеєра**) і почати

виконання спочатку – з першої інструкції на правильному шляху. Неправильне припущення призводить до значного зниження продуктивності, однак правильне вгадування напрямку розгалуження забезпечує виконання без затримок. Деякі процесори, у яких реалізовано прогнозування розгалуження, відстежують результати оцінок умови розгалуження та використовують цю інформацію для підвищення точності майбутніх прогнозів під час повторного виконання тієї самої інструкції.

- У разі виявлення інструкції умовного розгалуження процесор може почати виконання інструкцій по обох шляхах розгалуження одночасно, використовуючи свої суперскалярні можливості. Цей підхід називається **попереджувальним виконанням**. Після визначення умови розгалуження результати виконання по неправильному шляху відкидаються. Упереджувальне виконання може застосовуватися тільки в тому випадку, якщо інструкції не вносять змін, які не можна скасувати. Запис даних в основну пам'ять або на пристрій виводу є прикладом зміни, яку нелегко скасувати, тому випереджувальне виконання буде призупинено, якщо така дія відбудеться під час виконання неправильним шляхом.

Сучасні високопродуктивні процесори виділяють значну частину своїх логічних ресурсів для підтримки ефективної конвеєризації в найрізноманітніших умовах обробки. Сукупні переваги багатоядерної, суперскалярної та суперконвеєрної обробки забезпечують вражаюче підвищення продуктивності в останніх поколіннях складних процесорних архітектур. Реалізуючи конвеєрну обробку інструкцій і виконуючи інструкції в суперскалярному контексті, ці функції привносять паралелізм у послідовності коду, спочатку не призначені для паралельного виконання.

Продуктивність обробки можна додатково підвищити шляхом введення паралельного виконання декількох потоків на одному ядрі процесора з використанням одночасної багатопоточності.

Одночасна багатопоточність

Як ми дізналися з попередніх розділів, кожен виконуваний процес містить один або кілька потоків виконання. Під час реалізації багатопотоковості з розподілом за часом на одноядерному процесорі в будь-який момент часу в стані виконання перебуває тільки один потік. Швидко перемикаючись між кількома готовими до запуску потоками, процесор створює ілюзію (з погляду користувача) одночасного виконання кількох програм.

У цьому розділі представлено концепцію суперскалярного оброблення, що надає одному процесорному ядру можливість видавати більше однієї інструкцію за такт. Підвищення продуктивності в результаті суперскалярної обробки може бути обмежене, коли активна послідовність інструкцій не вимагає використовувати

поєднання ресурсів процесора, яке добре узгоджується з можливостями його суперскалярних функціональних блоків. Наприклад, у конкретній послідовності інструкцій можуть інтенсивно використовуватися блоки цілочисельних обчислень (що призводить до появи конвеєрних бульбашок), у той час як блоки обчислення адрес в основному простоюють.

Одним зі способів збільшити використання суперскалярних можливостей процесора є видача в кожному такті інструкцій з більш ніж одного потоку. Такий підхід називають **одночасною багатопотоковістю**. Одночасне виконання інструкцій з різних потоків підвищує ймовірність того, що послідовності інструкцій залежатимуть від несумісних функціональних можливостей процесора, тим самим забезпечуючи підвищений паралелізм виконання і вищу швидкість виконання інструкцій.

Процесори, що підтримують одночасну багатопотоковість, повинні надавати окремий набір регістрів для кожного одночасно виконуваного потоку, а також повний набір перейменованих внутрішніх регістрів для кожного потоку. Мета тут полягає в тому, щоб надати більше можливостей для використання суперскалярних функцій процесора.

Багато сучасних високопродуктивних процесорів підтримують виконання двох одночасних потоків, хоча деякі забезпечують підтримку до восьми таких потоків. Як і у випадках з більшістю інших методів підвищення продуктивності, що обговорюються в цій главі, збільшення кількості одночасних потоків, підтримуваних ядром процесора, в остаточному підсумку досягає точки падіння ефективності, тому що одночасні потоки конкурують за доступ до загальних ресурсів.

ОДНОЧАСНА БАГАТОПОТОЧНІСТЬ, БАГАТОДІЙНІ ПРОЦЕСОРИ ТА БАГАТОПРОЦЕСОРНІСТЬ



Поняття одночасної багатопоточності відноситься до ядра процесора, здатного підтримувати виконання інструкцій з різних потоків на одному етапі конвеєра в рамках одного тактового циклу. У багатоядерному процесорі, зі свого боку, кожне з кількох процесорних ядер, розміщених на одному кремнієвому кристалі, виконує інструкції незалежно від інших ядер і підключається до них тільки через кеш-пам’ять певного рівня.

Багатопроцесорний комп’ютер містить кілька інтегральних схем процесора, кожна з яких зазвичай входить до складу окремого пакета інтегральних схем. Як альтернативу багатопроцесорний комп’ютер можна реалізувати у вигляді декількох мікросхем процесорів, зібраних в одному корпусі.

Методи оптимізації, що обговорювалися досі, спрямовані на підвищення продуктивності скалярної обробки даних. Це означає, що кожна послідовність

інструкцій працює з невеликим числом значень даних. Навіть суперскалярне опрацювання, реалізоване в поєднанні з одночасною багатопоточністю або без неї, спрямоване на прискорення виконання інструкцій, які зазвичай у кожний момент часу працюють з одним або двома елементами даних розміром із регістр.

При обробці векторних даних одна й та сама математична операція виконується над кожним елементом масиву значень даних одночасно. Особливості архітектури процесора, що поліпшують паралелізм виконання операцій векторної обробки, розглядаються в наступному розділі.

Модель обробки SIMD

Процесори, які видають за такт одну інструкцію без елементів даних або з одним чи двома елементами даних, називають скалярними процесорами. Процесори, які здатні виконувати кілька інструкцій за такт, але не виконують у явному вигляді інструкції векторної обробки, називаються **суперскалярними процесорами**. Деякі алгоритми виграють від підходу до виконання, явно орієнтованого на векторне опрацювання, тобто від одночасного виконання однієї й тієї самої операції над багатьма елементами даних. Інструкції процесора, адаптовані до таких завдань, називають інструкціями, що реалізують модель обробки **з одним потоком інструкцій і безліччю потоків даних** (single instruction, multiple data, SIMD).

Одночасно видавані інструкції в суперскалярних процесорах зазвичай розв’язують різні завдання з різними даними, представляючи модель паралельного оброблення **з безліччю потоків інструкцій і безліччю потоків даних** (mul-tiple instruction, multiple data, MIMD). Деякі операції оброблення, як-от операція скалярного добутку, що використовується в цифровому обробленні сигналів, описана в *розділі 6*, виконують одну й ту саму математичну операцію над масою значень.

Операція **множення з накопиченням** (multiply accumulate, MAC), також описана в *розділі 6*, послідовно виконує скалярні математичні операції над кожною парою елементів вектора. Також можна розробити схемні рішення процесорів та інструкції, здатні виконувати аналогічні операції більш ніж над однією парою чисел одночасно.

У сучасних процесорах передбачено SIMD-інструкції для виконання таких завдань, як математичні операції з числовими масивами, оброблення графічних даних і пошук під рядків у символьних рядках.

Розроблені компанією Intel засоби розширення потокового опрацювання на основі моделі SIMD – **Streaming SIMD Extensions (SSE),** впроваджені в процесорах Pentium III 1999 р., являють собою набір інструкцій і засобів виконання для одночасного опрацювання 128-бітних масивів даних. Дані, що містяться в масиві, можуть складатися з цілих чисел або значень із плаваючою комою. Інструкції SSE другого покоління (**SSE2**) можуть паралельно обробляти дані таких типів:

- два 64-бітних значення з плаваючою комою;
- чотири 32-бітних значення з плаваючою комою;
- два 64-бітних цілих значення;
- чотири 32-бітних цілих значення;
- вісім 16-бітних цілих значень;
- шістнадцять 8-бітних цілих значень.

SSE2 надає інструкції обчислень із плаваючою комою для знайомих математичних операцій додавання, віднімання, множення і ділення.

Також доступні інструкції для обчислення квадратного кореня, зворотної величини, зворотної величини квадратного кореня і повернення максимального значення серед елементів масиву. Інструкції цілочисельних обчислень із SSE2 реалізують операції порівняння, побітові операції, перетасовування даних і перетворення типів даних.

У пізніших поколіннях набору інструкцій SSE збільшено розмір даних і різноманітність підтримуваних операцій. Остання ітерація можливостей типу SSE (станом на 2021 р.) знайшла вираження у вигляді набору інструкцій для вдосконалених векторних обчислень AVX-512. **AVX** розшифровується як **advanced vector extensions** і забезпечує підтримку регістрів розрядністю 512 біт. До складу AVX-512 включено, серед іншого, інструкції, оптимізовані для підтримки алгоритмів нейронних мереж.

Одна з перешкод на шляху широкого впровадження різних поколінь інструкцій SSE і AVX полягає в тому, що для їхнього застосування кінцевими користувачами ці інструкції мають бути реалізовані в процесорі, операційна система повинна підтримувати їх, а компілятори та інші аналітичні інструменти для користувачів повинні використовувати можливості SSE. Історично так склалося, що після введення нових інструкцій процесора були потрібні роки, перш ніж кінцеві користувачі могли легко використати всі їхні переваги.

SIMD-інструкції, доступні в сучасних процесорах, мабуть, знайшли своє наймасштабніше застосування в галузі наукових обчислень. Дослідникам, які працюють з комплексним моделюванням, алгоритмами машинного навчання або складним математичним аналізом, наявність інструкцій SSE і AVX дає можливість значно підвищити продуктивність коду, що виконує великі математичні операції, маніпуляції з символами та інші векторно-орієнтовані завдання.

Резюме

Більшість сучасних 32- і 64-розрядних процесорів поєднують у собі значну частину або навіть усі методи підвищення продуктивності, подані в цьому розділі. Типовий персональний комп'ютер або смартфон споживчого класу містить єдину інтегральну

схему основного процесора з чотирма ядрами, кожне з яких підтримує одночасну багатопотоковість із двома потоками. Цей процесор є суперскалярним, суперконвеєрним і використовує три рівні кеш-пам'яті. Процесор декодує інструкції в мікрооперації та виконує складне прогнозування розгалуження.

Методи, представлені в цьому розділі, можуть здатися надто складними і важкими для розуміння, проте в реальності кожен із нас регулярно застосовує їх і отримує реалізовані ними переваги в продуктивності щоразу, коли ми користуємося обчислювальними пристроями будь-якого типу. Логіка опрацювання, необхідна для впровадження конвеєризації та суперскалярних операцій, безсумнівно, складна, але виробники напівпровідників докладають зусиль для реалізації цих функцій з однієї простої причини: це окупається продуктивністю їх виробів і підсумковою цінністю цих продуктів у сприйнятті їх клієнтів.

У цьому розділі було представлено основні методи підвищення продуктивності, що використовуються в процесорах і комп'ютерних архітектурах для досягнення максимальної швидкості виконання в реальних обчислювальних системах. Ці методи жодним чином не змінюють те, що процесор видає на виході; вони просто допомагають швидше виконувати його роботу. Ми розглянули найважливіші методи підвищення продуктивності систем, включно з використанням кеш-пам'яті, конвеєрним опрацюванням інструкцій, одночасною багатопотоковістю і моделлю опрацювання SIMD.

Наступний розділ присвячено розширенням, які зазвичай реалізують на рівні набору інструкцій процесора з метою надання системам додаткових можливостей, окрім загальних вимог до обробки даних.

Серед розширень, що обговорюються в *розділі 9*, – привілейовані режими процесора, арифметика з плаваючою комою, керування живленням і керування безпекою системи.

Вправи

1. Розглянемо I-кеш L1 із прямим відображенням розміром 32 Кбайт. Кожен рядок кеша складається з 64 байт, а системний адресний простір становить 4 Гбайт. Скільки бітів відведено для тегу цього кеша? Біти з якими номерами (біт 0 – найменш значущий біт) складають адресне слово?
2. Розглянемо 8-канальний набірно-асоціативний кеш інструкцій і даних L2 об’ємом 256 Кбайт із 64 байтами в кожному рядку кеша. Скільки наборів використовується в цьому кеші?
3. Процесор має 4-етапний конвеєр з максимальними затримками 0,8; 0,4; 0,6 і 0,3 нс на етапах 1–4 відповідно. Якщо перший етап замінити двома етапами з

максимальними затримками 0,5 і 0,3 нс, відповідно, наскільки збільшиться тактова частота процесора у відсотковому вираженні?

9

Спеціалізовані розширення процесорів

У попередніх розділах ми обговорювали особливості обчислювальних архітектур загального призначення та деякі архітектурні спеціалізації, призначені для врахування вимог певних предметних областей. Цей розділ присвячено розширенням, які зазвичай реалізують на рівні набору інструкцій процесора та в інших апаратних засобах обчислювальної системи з метою надання додаткових можливостей, крім загальних вимог до опрацювання даних.

Прочитавши цей розділ, ви отримаєте уявлення про призначення привілейованих режимів процесора й особливості їхньої роботи в багатопроцесорних і багатокористувацьких контекстах. Ви ознайомитеся з концепціями процесорів і наборів інструкцій для обчислень із плаваючою комою, методами керування живленням у пристроях з акумуляторним живленням, а також із функціями процесора та системи, що підвищують рівень безпеки.

У цьому розділі ми обговоримо такі розширення процесора і системні функції:

- привілейовані режими процесора;
- арифметика з плаваючою комою;
- керування живленням;
- керування безпекою системи.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-OrgяnizaйonSecond-Edition.**

Привілейовані режими процесора

Більшість операційних систем, що працюють на 32- і 64-розрядних процесорах, керують доступом до системних ресурсів, використовуючи концепцію рівнів привілеїв. Основними причинами керування доступом у такий спосіб є підвищення стабільності системи, запобігання несанкціонованій взаємодії із системними апаратними засобами та запобігання несанкціонованому доступу до даних.

Привілейоване виконання підвищує стабільність системи, гарантуючи, що тільки довіреному коду дозволено виконувати інструкції, які забезпечують необмежений доступ до ресурсів, таких як регістри конфігурації процесора і пристрої введення-виведення. Ядро операційної системи і пов'язані з ним модулі, включно з драйверами пристроїв, вимагають привілейованого доступу для виконання своїх функцій. Оскільки будь-який збій процесу ядра або драйвера пристрою, найімовірніше, призведе до негайної зупинки роботи всієї системи, перед випуском для загального використання ці програмні компоненти зазвичай проходять процес ретельного проєктування і піддаються суворому тестуванню.

Запуск операційної системи в привілейованому контексті запобігає доступу неавторизованих додатків до керованих системою структур даних, таких як таблиці сторінок і таблиці векторів переривань. Випадково або внаслідок злого наміру користувацький застосунок може спробувати одержати доступ до даних, що містяться в системній пам'яті або в пам'яті, яка належить іншому користувачеві. Механізми керування доступом до системи запобігають цим спробам та інформують неправильно працюючий застосунок про порушення правил, ініціюючи виняток, що зазвичай призводить до завершення роботи програми з повідомленням про помилку.

У *розділі 3* було представлено концепції, пов’язані з перериваннями та їх обробленням процесором. Перш ніж заглибитися в деталі привілейованих режимів процесора, ми обговоримо оброблення переривань і винятків докладніше.

Обробка переривань і винятків

Апаратні переривання, як ми бачили, дають змогу процесорам швидко реагувати на запити обслуговування від периферійних пристроїв. Апаратне переривання сповіщає процесор про необхідність здійснити яку-небудь дію, зазвичай пов’язану з передачею даних на зовнішній пристрій або від нього.

Винятки аналогічні перериванням з тією лише різницею, що винятки зазвичай пов'язані з реакцією на деякі умови, що виникають усередині процесора. Одним із прикладів внутрішнього виключення є ділення на нуль.

Код користувацького рівня може навмисно генерувати винятки. На практиці це стандартний метод, який використовується непривілейованим кодом для запиту

системних сервісів, що надаються привілейованим кодом у ядрі операційної системи та драйверах пристроїв.

Неможливо точно сформулювати відмінність між термінами **"переривання"** і **"виняток"**. Обробка переривань і обробка винятків зазвичай залежать від одних і тих самих або подібних апаратних ресурсів процесора, які працюють практично однаково. В архітектурі x86 інструкція, яка ініціює програмно-генероване переривання (або виняток), позначається як *int* – скорочення від *interrupt* (переривання).

Винятки можуть виникати внаслідок помилок, що виникають під час виконання програм, таких як ділення на нуль, а також у ситуаціях, не пов’язаних із помилками, таких як відмови сторінок. Якщо переривання або виняток не призводять до радикальної реакції у вигляді припинення роботи додатка, процедура обслуговування переривання або обробник винятків обробляє подію і повертає керування системному планувальнику, який зрештою відновлює виконання перерваного потоку.

У разі виникнення переривання або винятку процесор передає керування відповідному оброблювачу в такий спосіб.

Коли відбувається *переривання*, процесор дозволяє завершити виконання поточної інструкції, а потім передає керування **процедурі обслуговування переривання** (intemipt service routine, ISR). Код в ISR здійснює обробку, необхідну для виконання запиту, ініційованого перериванням. Після завершення ISR виконання перерваного коду поновлюється. Ці дії виконуються без відома або підтвердження з боку будь-яких потоків, які можливо, виконувалися під час переривання.

Під час реагування на *виняток* процесор передає керування процедурі обробки винятків, яка багато в чому схожа на ISR. Під час виконання інструкції може виникнути виняток, що перешкоджає засвідченню операцій інструкції. Якщо виконання інструкції переривається через виняток, ту саму інструкцію буде перезапущено після завершення роботи оброблювача винятків і відновлення виконання потоку. Цей механізм допускає виникнення відмов сторінок у будь-який момент під час виконання програми без впливу на результати, отримані програмою (окрім введення затримки через обробку відмови сторінки).

Кожен тип переривання або виключення має номер вектора, що посилається на рядок у таблиці векторів системних переривань. У разі виникнення переривання або виключення апаратні засоби процесора звертаються до **таблиці векторів переривань** (intemipt vector table, IVT) для вибору відповідної процедури обслуговування переривання (ISR) або оброблення винятків. Рядок таблиці векторів переривань, що відповідає номеру вектора, містить адресу процедури обслуговування переривання або обробника для цього вектора.

Під час опрацювання переривання або винятку процесор поміщає будь-яку необхідну контекстну інформацію в стек і передає керування обробнику. У разі переривання

обробник обслуговує пристрій і видаляє його запит на переривання. Після опрацювання переривання або виключення обробник повертає керування операційній системі. У разі виключення це може спричинити перезапуск інструкції, яка виконувалася в момент виникнення виключення.

У табл. 9.1 наведено деякі типи переривань і винятків, що використовуються в процесорах x86, які працюють у захищеному режимі.

Таблиця 9.1. *Приклад типів переривань і виключень у x86*

Вектор	Опис	Причина
0	Помилка ділення	Інструкція `div` або `idiv` зробила спробу цілочисельного ділення на нуль
2	Немасковане переривання	Було отримано апаратний сигнал NMI
3	Контрольна точка	Виконано інструкцію `int 3`
6	Неприпустимий код операції	Було зроблено спробу виконати зарезервований код операції
13	Загальний захист	Була зроблена спроба забороненого доступу до пам'яті або системного ресурсу
14	Відмова сторінки	MMU видав запит на трансляцію віртуальної адреси
32 – 255	Визначаються користувачем	Ці вектори доступні для використання апаратними перериваннями або за допомогою інструкції `int`

Зазначимо деякі цікаві деталі, пов'язані з перериваннями і виключеннями, зазначеними в табл. 9.1.

- Робота вхідного сигналу NMI за вектором 2 аналогічна входу NMI процесора 6502, розглянутому в *розділі 3*.
- Сучасні процесори надають розвинуті можливості використання контрольних точок, які не порушують роботу системи, однак архітектура *x*86 зберігає функціональність контрольної точки, що надається інструкцією `int 3` з найперших днів існування процесора 8086. Як було показано в *розділі 3*, механізм, який використовують налагоджувачами програмного забезпечення процесора 6502 для переривання виконання програми за вказаною адресою, полягає в тимчасовій заміні коду операції в місці переривання на опкод `BRK`. Коли виконання програми досягає цього місця, обробник `BRK` перебирає на себе керування, щоб надати користувачеві можливість взаємодіяти із системою. Інструкція `int` з в архітектурі x86 функціонує таким самим чином.
- На відміну від інструкції `int` процесорів *x*86, використовуваної з будь-якими іншими номерами векторів, інструкція `int 3` реалізована у вигляді

однобайтового опкоду зі значенням `CCh`. Програмне переривання з іншим вектором, наприклад `int 32`, являє собою двобайтову інструкцію. Інструкція `int 3` дозволяє вставляти контрольну крапку шляхом заміни одного байта коду значенням `CCh`.

- Вектор 13, обробник винятків загального захисту, активується за будь-якої спроби програми отримати доступ до пам’яті або системного ресурсу, не виділеного для такого використання. У сучасних операційних системах системний обробник для цього вектора завершує роботу запущеного додатка і відображає повідомлення про помилку.
- Вектор 14, обробник відмов сторінок, спрацьовує, коли застосунок намагається отримати доступ до допустимої сторінки, яка відсутня у фізичній пам'яті. Цей обробник намагається знайти зазначену сторінку, яка може міститися у файлі на диску або в системному файлі підкачки, завантажує сторінку в пам’ять, оновлює таблиці трансляції віртуальних адрес у фізичні та перезапускає інструкцію, що спричинила виняток.

Підбиваючи підсумок, можна сказати, що апаратні переривання ініціюються пристроями вводу-виводу, які потребують передання даних або інших видів обслуговування. Винятки виникають, коли виконання послідовності інструкцій має бути призупинено для обробки такої умови, як відмова сторінки або спроба несанкціонованого доступу до пам’яті.

Апаратні переривання, як правило, відбуваються у випадкові моменти часу відносно поточного виконання коду програми, тоді як поведінка, пов’язана з виключеннями, генерованими програмно, часто повторюється, якщо той самий код виконують знову під час роботи з тими самими даними.

Деякі винятки, як-от помилка загального захисту, призводять до завершення процесу, що спричинив виняток, але більшість переривань і винятків завершуються відновленням виконання перерваного потоку після того, як обробник переривання або винятку завершив свою роботу.

ВИНЯТКИ В МОВАХ ПРОГРАМУВАННЯ



Багато мов програмування надають засоби для обробки винятків у додатках. Такі винятки істотно відрізняються від винятків, що обробляються на рівні процесора. Винятки в мовах програмування зазвичай пов'язані з помилками на значно вищому концептуальному рівні, ніж винятки, що обробляються апаратними засобами процесора.

Наприклад, C++ генерує (або «видає», в термінології C++) виняток при невдалому запиті на виділення пам'яті. Цей виняток зовсім

іншого типу в порівнянні з системними винятками, що обробляються на рівні процесора. Будьте уважні і не плутайте винятки в мовах програмування високого рівня з винятками, що обробляються безпосередньо процесором.

Кільця захисту

Стратегія захисту, що застосовується в сучасних процесорах і операційних системах, у чомусь схожа на систему оборони, що знайшла втілення в устрої середньовічних замків. Замок зазвичай має високу стіну, що оточує його територію, іноді доповнену ровом. У цій зовнішній стіні є невелика кількість добре захищених проходів, кожен з яких захищений від зловмисників такими механізмами, як розвідні мости і цілодобова охорона. Сам замок, розташований на огородженій стіною території, також має міцні стіни і невелику кількість добре захищених проходів, що ще більше обмежує доступ до найвразливіших місць.

Найпривілейованіші члени замкової ієрархії мають необмежені права на вхід і вихід із замку та прохід через зовнішню стіну. Менш привілейовані члени суспільства можуть мати дозвіл на прохід всередину через зовнішню стіну, але їм заборонений доступ безпосередньо в замок. Найменш привілейованим представникам місцевого населення здебільшого заборонено входити до замку, а в тих випадках, коли доступ їм дозволено (наприклад, у певні громадські зони), вони мають прийняти обмеження щодо того, що їм дозволено робити.

Захист, що забезпечується цією стратегією, можна представити у вигляді концентричних кілець, при цьому для доступу до внутрішніх кілець потрібні найвищі привілеї, а зовнішнє кільце являє собою сферу, що потребує найменших привілеїв, як показано на рис. 9.1.



Рис 9.1. *Приклад захисних кілець*

Ця стратегія захисту реалізує три рівні привілеїв, які визначають типи доступу для всіх користувачів системи. Кільце 0 вимагає найвищого рівня привілеїв, а кільце 2 не вимагає особливих привілеїв.

Сучасні процесори й операційні системи, що працюють на їхній основі, використовують дуже схожий підхід для запобігання несанкціонованому доступу до критичних ресурсів і надання непривілейованим користувачам доступу до можливостей системи, схвалених для їхнього використання.

В принципі, можна створити кілька проміжних рівнів між користувачами з найвищими та найнижчими привілеями, однак у більшості сучасних комп'ютерних систем реалізовано лише два кільця: привілейоване кільце, яке називають ядром або супервізором, і непривілейоване користувацьке кільце. У деяких операційних системах є проміжне кільце, яке містить драйвери пристроїв і надає доступ до ресурсів, необхідних для взаємодії з пристроями вводу-виводу, але не забезпечує безперешкодного загальносистемного доступу в кільце ядра.

Одна з причин, через яку операційні системи, такі як Windows і Linux, підтримують тільки два кільця привілеїв, полягає в тому, що під час проєктування цих систем ставилося завдання переносимості на різні процесорні архітектури. Деякі процесори підтримують тільки два кільця привілеїв, але є й інші, які підтримують більшу кількість кілець. Операційна система, яку можна перенести, не може реалізувати більше двох кілець привілеїв, якщо архітектура будь-якого з базових процесорів не підтримує бажану кількість кілець. Архітектура *x*86, наприклад, підтримує до чотирьох кілець, але в Windows і Linux використовуються тільки два з них (кільце 0 з найбільшими привілеями і кільце 3 з найменшими привілеями).

На рис. 8.2 представлено кільцеву організацію більшості операційних систем. тим, що працюють на архітектурі *x*86.



Рис 9.2. *Кільця захисту в операційних системах x86*

Кільцева система керування привілеями є основною причиною того, що сумнозвісний "синій екран смерті" Windows, що був поширений у 1990-х роках, так рідко з'являється у користувачів останніх версій Windows. Користувацькі додатки, такі як веб-браузери, поштові клієнти та текстові редактори, іноді стикаються з проблемами, що призводять до аварійного завершення їхньої роботи. Завдяки дворівневому механізму застосування привілеїв, що надається сучасними операційними системами, такими як Windows, Linux, macOS і Android, збиток від збою в роботі окремого додатка стримується операційною системою, не даючи йому змоги вплинути на саму операційну систему або будь-яку іншу програму, запущену в системі (принаймні на ті програми, які не залежать від коректної роботи програми, в якій стався збій).

Після аварійного завершення роботи додатка операційна система очищає всі ресурси, використані програмою, що відмовила, включно з виділеними сторінками пам'яті та відкритими файлами. Завдяки цьому комп'ютерні системи, такі як веб-сервери, можуть безперервно працювати впродовж сотень днів, незважаючи на періодичні збої та перезапуски програмних додатків, що працюють на них.

Крім захисту від збоїв додатків, управління привілеями на основі кілець дає суттєві переваги в галузі захисту від зловмисників. Один із типів атак, які може спробувати зробити хакер, – вставити деякий код у системний модуль, що працює в кільці 0. Така вставка може відбутися у виконуваний файл на диску, або зловмисник може спробувати змінити код ядра під час його роботи в пам'яті. У разі успіху зловмисник може використовувати цей код для доступу до даних у будь-якій частині системи, оскільки він працює в кільці 0.

Успіх такої атаки не є чимось неможливим, проте в сучасних процесорах і операційних системах реалізовано великий набір заходів безпеки та усунуто багато вразливостей, які були присутні в ранніх версіях операційних систем. Коли системні адміністратори і користувачі повною мірою використовують переваги засобів захисту на основі кілець, доступних у сучасних комп'ютерних системах, у зловмисника залишається дуже мало можливих шляхів для отримання доступу до захищених даних. Дійсно, у більшості випадків ключовим елементом успішних зломів, про які повідомляють у наші дні в новинах, є порушення правил безпеки людиною, яке використовував у своїх цілях Зловмисник. Ми обговоримо кіберзагрози та методи гарантування безпеки системного та прикладного програмного забезпечення пізніше в *розд.* "Управління безпекою системи" цього розділу. *"Керування безпекою системи" цього розділу*, а також у *розділі 14*.

Режим супервізора і режим користувача

У дворівневій ієрархії захисту на основі кілець, рівень захисту потоку, що виконується, представлений бітом у реєстрі. Під час роботи в кільці 0, біт **режиму супервізора** має значення 1, а під час роботи в **режимі користувача** (кільце 3 у *x*86)

цей біт дорівнює 0. Біт режиму супервізора може бути змінений тільки кодом, що працює в режимі супервізора.

Стан цього біта визначає, які інструкції доступні для виконання потоком. Інструкції, які можуть завадити роботі системи, наприклад інструкція `hlt` у *x*86, яка зупиняє виконання інструкцій процесора, недопустима в режимі користувача.

Будь-яка спроба виконати заборонену інструкцію призводить до збою загального захисту. У режимі користувача доступ інструкцій до областей системної пам'яті та пам'яті, виділеної іншим користувачам, заборонений. У режимі супервізора всі інструкції доступні для виконання, і також доступні всі області пам'яті.

За аналогією із замком, біт режиму супервізора являє собою наданий охоронцям замку дозвіл, що відкриває доступ на територію замку і в сам замок. Встановлений біт режиму супервізора надає вам ключі від королівства.

Системні виклики

Весь код, що належить до ядра і драйверів пристроїв, завжди працює в кільці 0. Увесь користувацький код завжди виконується в кільці 3, навіть для користувачів із підвищеними привілеями в операційній системі, таких як системні адміністратори. Код, що виконується в кільці 3, суворо контролюється системою і не може безпосередньо виконувати будь-які дії, пов'язані з виділенням пам'яті, відкриттям файлу, виведенням інформації на екран або взаємодією з пристроєм введення-виведення. Для того щоб отримати доступ до будь-якої з цих системних функцій, користувацький код кільця 3 повинен відправити в ядро запит на обслуговування.

Цей запит на обслуговування повинен спочатку пройти через "ворота" (як відвідувачі, які входять до нашого замку), де тип запитуваної операції, а також усі пов'язані з нею параметри проходять ретельну перевірку і затвердження, перш ніж буде надано дозвіл на виконання операції. Код, що виконує запитану операцію, запускається в режимі супервізора в кільці 0 і після завершення повертає управління в режим користувача в кільці 3.

У ранніх версіях Windows (до Windows XP) для запиту системних послуг додаток використовував механізм програмного переривання з вектором `2eh`. Протокол виклику системного сервісу містив у собі поміщення в регістри процесора параметрів, необхідних для запитуваного сервісу, і виконання інструкції `int 2eh`, що ініціює програмне переривання. Обробник працював у режимі супервізора, що призводило до переходу від кільця 3 до кільця 0. Після завершення роботи обробника система поверталася до кільця 3 для виконання інструкції, наступної за `int 2eh`.

Одна з проблем, пов'язаних із використанням механізму `int 2eh` для запиту послуг ядра, полягає в його невисокій ефективності. Дійсно, потрібно понад 1000 тактових циклів процесора, щоб пройти шлях від точки, в якій виконується інструкція `int 2eh`,

до коду ядра, який починає обробляти це виключення. Завантажена система може запитувати послуги ядра тисячі разів на секунду.

Для розв'язання цієї проблеми Intel впровадила в архітектурі *x*86, починаючи з процесора Pentium II у 1997 р., інструкції `sysenter` і `sysexit`. Їхня мета полягала в прискоренні процесу переходу з кільця 3 у кільце 0, а потім повернення в кільце 3. Завдяки використанню цих інструкцій замість `int 2eh` процес переходу в режим ядра і повернення з нього прискорюється приблизно втричі.

Приблизно в той час, коли компанія Intel почала випускати процесори з інструкціями `sysenter` і `sysexit`, AMD випустила інструкції `syscall` і `sysret` у своїх процесорних архітектурах з тією ж метою підвищення продуктивності. На жаль, інструкції в процесорних архітектурах Intel і AMD несумісні, внаслідок чого до операційних систем висувається вимога розрізняти архітектури під час використання інструкцій для прискорення виклику сервісів ядра.

У наступному розділі ми розглянемо формати даних і операції, пов'язані з арифметикою з плаваючою комою.

Арифметика з плаваючою комою

Сучасні процесори зазвичай підтримують цілочисельні типи даних розрядністю 8, 16, 32 і 64 біта. Деякі невеликі вбудовані процесори можуть не підтримувати 64-бітові або навіть 32-бітові цілі числа безпосередньо, в той час як більш складні пристрої здатні підтримувати 128-бітові цілі числа. Цілочисельні типи даних підходять для використання в широкому спектрі додатків, але багато галузей обчислень, особливо у сфері науки, інженерної справи та навігації, вимагають можливості з високим ступенем точності представляти дробові числа.

Як простий приклад обмежень цілочисельної математики припустимо, що вам потрібно розділити 5 на 3. На комп'ютері, обмеженому використанням тільки цілих чисел, ви можете виконати цілочисельне обчислення цього виразу, використовуючи наступний код мовою C++:

```
#include
int main(void)
{
    int a = 5;
    int b = 3;
    int c = a / b;
    std::cout << "c = " << c << std::endl;
    return 0;
}
```

Ця програма видає такий результат:

```
c = 1
```

Якщо ви введете цей вираз у кишеньковий калькулятор, то виявите, що результат, отриманий програмою, не надто близький до фактичного результату, який становить приблизно 1,6667. В обчислювальних додатках, де потрібні точні обчислення з дійсними числами, практичне рішення пропонує арифметика з плаваючою комою.

З погляду математики, множина дійсних чисел складається з усіх чисел, включно з усіма цілими і дробовими числами, розташованими на числовій прямій від від'ємної нескінченності до позитивної нескінченності. Кількість цифр у цілій і дробовій частинах дійсного числа не обмежена.

З огляду на обмежений обсяг пам'яті, доступний навіть у найбільшому комп'ютері, очевидно, що в комп'ютерних програмах неможливо уявити весь діапазон дійсних чисел. Якщо ми хочемо представляти дійсні числа корисним чином, необхідний компроміс. Стандартним рішенням є представлення дійсних чисел за допомогою мантиси і порядку.

У галузях знань, пов'язаних із математикою дуже великих чи дуже малих чисел, заведено представляти такі числа у вигляді мантиси і порядку. Наприклад, у фізиці гравітаційна стала представлена в такому форматі:

$$G = 6{,}674 * 10^{-11} \frac{\text{м}^3}{\text{кг} * \text{с}^2}.$$

Мантиса являє собою ненульові цифри числа після множення на мас-штабний коефіцієнт, який поміщає ці цифри в зручний діапазон. **Порядок** являє собою масштабний коефіцієнт, на який необхідно помножити мантису для отримання фактичного значення.

У цьому прикладі мантиса становить 6,674, а порядок дорівнює -11. У цьому прикладі використовується десяткова мантиса, що зручно для ручних розрахунків, оскільки множення на масштабний коефіцієнт 10^{-11} можна виконати, просто перемістивши положення десяткової коми мантиси. У цьому прикладі порядок -11 вимагає переміщення десяткової коми на 11 місць ліворуч, що дає еквівалентне значення G = 0,00000000006674.

Застосування нотації з плаваючою комою дає змогу уникнути помилок, пов'язаних із використанням довгих послідовностей нулів, і дає можливість представляти як дуже великі, так і дуже малі числа в компактному форматі.

У нотації з плаваючою комою можна подати будь-яке число на лінії дійсних чисел. Однак, якщо метою є подання всіх дійсних чисел, не повинно бути обмежень щодо кількості цифр у мантисі або порядку.

У комп'ютерних поданнях чисел із плаваючою комою і мантиса, і порядок обмежені наперед визначеною шириною бітового поля. Ці діапазони вибираються таким чином,

щоб числа з плаваючою комою вкладалися в стандартну ширину даних, зазвичай 32 або 64 біти, забезпечуючи при цьому достатню кількість розрядів мантиси і порядку для широкого спектра практичних завдань.

Збільшення числа розрядів мантиси підвищує точність числових значень, вироблених у форматі з плаваючою комою. Збільшення числа цифр порядку збільшує діапазон чисел, які можуть бути представлені. Через кінцеву довжину мантиси і порядку результат обчислень із плаваючою комою часто не відповідає реальному результату. Замість цього найкращий результат у такій ситуації полягає в тому, що результат із плаваючою комою буде найближчим можливим значенням до правильного результату.

Сучасні процесори зазвичай підтримують 32- і 64-бітове подання чисел із плаваючою комою. Замість десяткового порядку, описаного в попередньому прикладі, комп'ютери працюють із порядком за основою 2. Загальний формат комп'ютерного числа з плаваючою комою:

$$\text{Знак} * \text{мантиса} * 2^{\text{порядок}}.$$

Знак – це просто +1 або -1. У двійковому поданні додатне число має знаковий біт, що дорівнює 0, а від'ємне число – знаковий біт, що дорівнює 1. Якщо відокремити знак від іншої частини числа, значення, що залишилося, є невід'ємним. Для будь-якого значення, відмінного від нуля (який розглядається як особливий випадок), це число можна привести в діапазон від 1 (включно) до менш як 2 шляхом множення на деякий ступінь 2.

Продовжуючи приклад із гравітаційною постійною, оскільки знак дорівнює +1, значення після видалення знаку залишається незмінним: $6{,}674 - 10^{-11}$. Мантису цього числа можна привести в діапазон ($1 \leq m \leq 2$), де m являє собою мантису, помножену на 2^{34} . Результат цього масштабування:

$$G = +1 * 1{,}1465845 * 2^{-34}.$$

Для того щоб мантиса потрапила в потрібний діапазон, ми помножили вихідне число на 2^{34} , тому щоб компенсувати операцію масштабування, подання числа з плаваючою комою необхідно помножити на 2^{-34}.

Оскільки наші комп'ютери працюють із двійковими числами, ми повинні перетворити мантису в двійкове представлення. Формат, який використовується під час обробки з плаваючою комою, повинен представляти число з діапазону від 1 до 2 у вигляді цілого числа без знаку.

Наприклад, якщо ми припустимо, що двійкова мантиса має розрядність 16 біт, то мантиса 1,0 буде представлена числом `0000h`, а значення, що трохи менше за 2,0 (фактично, 1,99998474) - числом `FFFFh`. Десяткова мантиса m перетворюється у двійкову 16-бітну мантису за допомогою виразу $(m - 1) * 2^{16}$ та округлення результату до найближчого цілого числа. Двійкова 16-бітна мантиса m перетворюється в десяткову за допомогою виразу $1 + m * 2^{-16}$.

У нашому прикладі десяткова мантиса 1,1465845 перетворюється на 16-бітну двійкову. двійкову мантису виду `0010010110000111b` або `2587h`.

За допомогою процедури масштабування, описаної в цьому розділі, числа з плаваючою комою можна подати в двійковій формі з будь-якою бажаною розрядністю для мантиси і порядку.

Для цілей сумісності між процесорами корисно визначити обмежену кількість двійкових форматів із плаваючою комою, здатних підтримувати широкий спектр варіантів застосування, і прийняти ці формати в масштабах усієї галузі.

З цієї причини в 1985 р. було ухвалено стандарт IEEE 754, що визначає комп'ютерне подання чисел із плаваючою комою. Перш ніж ознайомитися з основними положеннями цього стандарту, розглянемо конструкцію співпроцесора Intel 8087, призначеного для обчислень із плаваючою комою, що став джерелом багатьох концепцій, пізніше закріплених у стандарті IEEE 754.

Співпроцесор 8087 для обчислень з плаваючою комою

Сучасні процесори з апаратною підтримкою обчислень із плаваючою комою зазвичай реалізують набір відповідних інструкцій і виконують ці операції у виділеному функціональному блоці. У суперскалярному процесорі ядро головного процесора виконує інструкції інших категорій, а математичний співпроцесор (floating point unit, FPU) паралельно виконує обчислення з плаваючою комою.

Згадайте *розділ 1*, там ішлося про те, що оригінальний IBM PC 1981 р. містив гніздо для співпроцесора 8087 з плаваючою комою. Апаратні засоби співпроцесора 8087 виконують обчислення з плаваючою комою зі швидкістю приблизно в 100 разів вищою, ніж функціонально еквівалентна йому програмна реалізація, що працює на головному процесорі.

Оскільки встановлення 8087 було необов'язковим, більшість програм для ПК, що хотіли скористатися його можливостями, спочатку перевіряли наявність 8087 і в разі його відсутності поверталися до бібліотеки набагато повільнішого коду для обчислень із плаваючою комою.

Співпроцесор 8087 підтримує такі типи даних для обробки чисел:

- 16-бітне ціле число в додатковому коді;
- 32-бітове ціле число в додатковому коді;
- 64-бітове ціле число в додатковому коді;
- 18-значне упаковане двійково-десятнчне число (binary coded decimal, BCD) зі знаком;

- 32-бітове коротке дійсне число зі знаком, 24-бітною мантисою і 8-бітовим порядком;
- 64-бітове довге дійсне число зі знаком, 53-бітною мантисою та 11-бітним порядком;
- 80-бітове тимчасове дійсне число зі знаком, 64-бітною мантисою і 15-бітним порядком.

Дані кожного типу зберігаються в пам'яті у вигляді послідовності байтів. Формати, що використовуються для типів даних, які представляють дійсні числа, показано на рис. 9.3.



Рис 9.3. *Формати даних із плаваючою комою в співпроцесорі 8087*

Формати коротких і довгих дійсних чисел використовують припущений біт 1 як старший біт мантиси і не включають цей біт у свої двійкові представлення. Як особливий випадок нуль представляється в цих форматах шляхом встановлення нульових значень для мантиси і порядку.

Формат тимчасового дійсного числа використовується всередині співпроцесора 8087 для зберігання проміжних результатів. Цей формат має збільшену точність порівняно з форматом довгого дійсного числа, щоб звести до мінімуму розповсюдження помилок округлення по серії обчислень.

Кожен формат дійсних чисел можна представити у вигляді

$$(-1)^S(2^{E-\text{зміщення}})(m),$$

де S – це біт знаку, E – порядок, а m – мантиса. *Зміщення* дорівнює 127 для формату коротких дійсних чисел, 1023 для формату довгих дійсних чисел і 16 383 для формату тимчасових дійсних чисел.

Віднімання *зміщення* перетворює беззнакове ціле число, що зберігається в полі рядка, на знакове значення.

У нашому прикладі дійсне число G з десятковою мантисою 1,1465845 і двійковим масштабом 2^{34} представлено у форматі короткого дійсного числа зі знаковим бітом, що дорівнює нулю, порядком (-34 + 127) = 5Dh і мантисою (1,1465845 - 1) * 2^{23} = 12C348h. Комбінуючи всі три компоненти, одержуємо таке 32-бітове подання значення:

$$6{,}674 * 10^{-11} : (\text{біт знаку} = 0) * 2^{31} + \textit{зміщений порядок} =$$

$$= 5\text{Dh} * 2^{23} + (\text{мантиса} = 12\text{C348h}).$$

Підсумкове 32-бітове значення з плаваючою комою одинарної точності: `2E92C348h`.

Співпроцесор 8087 додає до набору інструкцій 8086/8088 68 позначень опкодів для виконання арифметичних, тригонометричних, експоненціальних і логарифмічних функцій. У програмі для ПК, що використовує співпроцесор 8087, код складається з одного потоку інструкцій 8088 і 8087, які витягуються звичайним послідовним способом процесором 8088. Сопроцесор 8087 пасивно контролює шини адрес і даних у міру виконання процесором інструкцій і підключається тільки при появі опкоду 8087. Процесор 8088 розглядає інструкції 8087 як інструкції без операції (або пор) та ігнорує їх.

Коли 8087 починає виконувати інструкцію, він може взяти під контроль шину хоста для передавання даних між своїми внутрішніми регістрами та системною пам’яттю за допомогою циклів прямого доступу до пам’яті (DMA). 8087 і 8088 не передають дані між собою безпосередньо і можуть обмінюватися даними, тільки зберігаючи їх у пам’яті для використання іншим процесором.

Виконання інструкцій співпроцесора 8087 відбувається незалежно від 8088, що робить ПК, оснащений 8087, по-справжньому паралельною системою обробки. У 8087 передбачено вихідний сигнал `BUSY`, що дає змогу процесору 8088 визначити, чи обробляє 8087 у цей момент інструкцію.

Коли процесору 8088 потрібні результати роботи 8087, 8088 повинен почекати вимкнення сигналу `BUSY`, і в цей момент він отримує доступ до осередків пам’яті, що містять результат інструкції 8087.

Стандарт обчислень з плаваючою комою IEEE 754

Найбільш широко використовуваними форматами представлення чисел із плаваючою комою в сучасних комп’ютерних системах є формати, визначені в стандарті IEEE 754. **Інститут інженерів з електротехніки та електроніки** (Institute of Electrical and Electronic Engineers, IEEE) публікує широкий спектр стандартів, пов’язаних з електротехнікою, електронікою та обчислювальною технікою. Початкова версія

стандарту IEEE 754 була прийнята в 1985 р. і значною мірою ґрунтувалася на типах даних і математичних операціях спів-процесора Intel 8087.

8087 не повністю відповідав початковій версії стандарту IEEE 754, який було опубліковано через кілька років після появи 8087. Пізніші співпроцесори Intel із плаваючою комою, починаючи з 80387 у 1987 р., уже повністю відповідали цьому стандарту. У сучасних 32- і 64-розрядних процесорах блок обчислень із плаваючою комою, що відповідає стандарту IEEE 754, зазвичай розміщують на тій самій інтегральній мікросхемі, що й основний процесор.

Стандарт IEEE 754 було оновлено у 2008 р. і ще раз – у 2019 р. Поточна версія IEEE 754- 2019 містить визначення для форматів чисел із плаваючою комою розрядністю 16, 32, 64, 128 і 256 біт. Ця версія також містить формати десяткових чисел із плаваючою комою розрядністю 32, 64 і 128 біт. 32-бітний і 64-бітний формати з плаваючою комою для двійкових чисел зазвичай підтримуються в мовах програмування, що включають операції з плаваючою комою. Підтримка інших типів даних IEEE 754, як правило, має більш обмежене поширення і не є стандартною для різних процесорів, мов програмування та операційних систем.

У наступному розділі представлено функції, реалізовані в багатьох сучасних процесорних архітектурах для керування енергоспоживанням системи.

Керування живленням

Для користувачів мобільних пристроїв з живленням від акумулятора, таких як смартфони, планшети та ноутбуки, можливість тривалої роботи без підзарядки є важливою характеристикою. Розробники мобільних пристроїв приділяють велику увагу тому, щоб витрата енергії акумулятора була мінімальною за будь-яких умов експлуатації.

Ось деякі методи, які використовують проектувальники для зниження енергоспоживання.

- Переведення непрацюючих підсистем у стан низького енергоспоживання або їхнє повне вимкнення, коли вони не потрібні. Застосування цього методу може виявитися неможливим для периферійних пристроїв, які мають бути доступні для реагування на вхідні запити, наприклад від мережевих інтерфейсів.
- Зниження напруги живлення інтегральних схем і тактових частот у періоди, коли швидкість виконання не критична.
- Коли це можливо, збереження інформації про стан системи та вимкнення живлення процесора. Користувачі мобільних комп'ютерів знайомі з двома режимами зниження енергоспоживання, коли система не використовується: режим очікування і сплячий режим. У режимі очікування живлення подається тільки на оперативну пам’ять системи, тоді як інша частина системи вимкнена.

Режим очікування забезпечує дуже швидкий запуск, коли користувач відновлює роботу з системою. За цю оперативність доводиться платити: для підтримки оперативної пам’яті в робочому стані потрібна значна кількість енергії. У сплячому режимі всі дані про стан системи записуються на диск, і система повністю вимикається. Сплячий режим не вимагає практично ніякого живлення, хоча для відновлення роботи зазвичай потрібно трохи більше часу, ніж у режимі очікування.

- Якщо потрібне періодичне опрацювання, наприклад у додатках реального часу, то під час завершення кожного етапу опрацювання процесор переводиться в стан низького енергоспоживання. Процесор залишається в цьому стані доти, доки переривання за таймером (або переривання іншого типу) не "розбудить" процесор для наступної ітерації. Багато вбудованих процесорів підтримують режим очікування з низьким енергоспоживанням, у якому конвеєр інструкцій зупиняється, але залишається готовим миттєво відновити виконання у відповідь на переривання. Деякі реалізації ОСРЧ підтримують цю концепцію, переводячи процесор у режим очікування, коли всі завдання блокуються в очікуванні переривання або іншої події.

Смартфони та інші пристрої з живленням від акумулятора широко використовують ці методи для мінімізації розряду акумулятора, зберігаючи готовність миттєво відреагувати на дії користувача і на зовнішні впливи, як-от вхідні виклики та повідомлення із соціальних мереж.

Сучасні процесори у високопродуктивних комп’ютерах і мобільних пристроях зазвичай підтримують можливість регулювання тактової частоти процесора під час виконання коду, а в деяких випадках можуть змінювати напругу живлення процесора. Це поєднання методів керування живленням розглядається в наступному розділі.

Динамічна зміна напруги і частоти

Оптимізація тактової частоти процесора і напруги живлення з метою звести енергоспоживання до мінімуму називається **динамічною зміною напруги і частоти** (dynamic voltage frequency scaling, DVFS). У режимі DVFS процесор регулярно оцінює вимоги свого поточного робочого стану до продуктивності та регулює тактову частоту і напругу живлення, щоб звести до мінімуму використання заряду акумулятора, продовжуючи при цьому працювати на прийнятному рівні.

У КМОН-чипі, що містить безліч транзисторів, розумна оцінка споживаної потужності становить:

$$P = \sum_{i=1}^{N} C_i v_i^2 f_i.$$

У цьому виразі P відображає загальне енергоспоживання чипа, який містить N МОН-транзисторів; C_i – ємність, створювана i-м транзистором; v_i – напруга живлення цього транзистора; f_i – його робоча частота.

Розглядаючи цю формулу, ми бачимо, що споживана потужність схеми пропорційна квадрату напруги живлення і прямо пропорційна ємності та робочій частоті. Оскільки ємність схеми відіграє значну роль в енергоспоживанні, вигідно виробляти пристрій за допомогою процесу виготовлення, який зменшує розмір вентилів схеми, тим самим зводячи ємність до мінімуму. Це одна з причин, через яку перехід на технологічний процес виробництва інтегральних схем, що підтримує менші розміри елементів, призводить до створення пристроїв зі зниженим енергоспоживанням.

Режим DVFS намагається звести до мінімуму загальне енергоспоживання, постійно знижуючи напругу живлення і тактову частоту процесора настільки, наскільки це можливо.

Зниження напруги живлення схеми може забезпечити значне зниження енергоспоживання. Однак будь-яке зниження напруги живлення процесора має бути ретельно узгоджено зі зміною тактової частоти. При зниженні напруги живлення КМОН-транзистори перемикаються повільніше. Це зумовлено тим, що ємність, створювана кожним транзистором, залишається незмінною, але її заряд і розряд виконуються повільніше, оскільки процес заряду/розряду відбувається при меншій напрузі. Для того щоб при зниженні напруги системи схема працювала правильно, необхідно одночасно знімати тактову частоту.

За аналогією з гідравлічною системою з *розділу 2*, ефект від зниження напруги живлення КМОН еквівалентний зниженню тиску води, що наповнює повітряну кульку, прикріплену до стінки труби: при зниженні тиску в системі кулька наповнюється повільніше.

Зниження напруги живлення процесора також знижує завадостійкість схеми. Коли це відбувається, зовнішнім перешкодам, таким як електричне поле, що збуджується під час запуску двигуна в побутовому приладі, стає легше порушити внутрішню роботу процесора. Будь-яке зниження напруги процесора необхідно ретельно контролювати для забезпечення його надійної роботи.

Зниження тактової частоти процесора (на додаток до будь-якого уповільнення тактової частоти, необхідного через зниження напруги живлення) знімає енергоспоживання приблизно пропорційно. Зниження робочої частоти також може підвищити надійність системи, оскільки для перемикання кожного вентиля потрібно більше часу на поширення і досягнення кінцевого стану, перш ніж він буде зафіксований вентилями, якими він керує.

Складний мобільний пристрій, як-от смартфон, часто потребує швидкого переходу між станами з високим і низьким енергоспоживанням під дією різних вхідних сигналів, зокрема дій користувача, подій, що запускаються за таймером, і

надходження інформації через бездротові з'єднання. Для системи вигідно швидко перемикатися між режимами високого та низького енергоспоживання в міру зміни обставин, однак також важливо обмежити частоту, з якою відбуваються такі переходи, оскільки сам акт перемикання між цими режимами вимагає певної витрати енергії.

Користувачі очікують, що їхні комп'ютери та інші складні цифрові пристрої споживатимуть не більше енергії, ніж необхідно, але вони також розраховують на те, що ці системи будуть безпечними. У наступному розділі представлено ключові аспекти управління безпекою системи.

Керування безпекою системи

Ми бачили, як поділ рівнів привілеїв між режимами ядра і користувача підтримує ефективний поділ додатків, запущених одним користувачем, від додатків інших користувачів і від системних процесів. Це забезпечує безпеку на рівні виконуваних програм.

В цілому це добре, але як бути з системами, які повинні залишатися безпечними навіть тоді, коли до них мають необмежений фізичний доступ звичайні неперевірені користувачі? Для того щоб запобігти доступу недосвідчених користувачів або зловмисників до захищеного коду, даних і апаратних ресурсів, потрібно реалізувати додаткові заходи на апаратному рівні.

Перш ніж перейти до детального опису засобів захисту на апаратному рівні, корисно перерахувати деякі категорії інформації та інших ресурсів, які необхідно захищати в цифрових системах.

- **Особиста інформація.** Така інформація, як державні ідентифікаційні номери, паролі для доступу до банківських рахунків, списки контактів, електронні листи та текстові повідомлення, має бути захищена навіть у разі втрати або крадіжки мобільного пристрою, що містить цю інформацію.
- **Ділова інформація.** Торгові секрети, списки клієнтів, описи експериментальних продуктів і стратегічні плани – ось деякі категорії конфіденційних бізнес-даних, що можуть мати велику цінність у руках конкурентів або злочинців. Підприємства також збирають велику кількість особистої інформації про своїх клієнтів і зобов'язані докладати значних зусиль для забезпечення безпеки цієї інформації.
- **Державна інформація.** Державні організації зберігають величезну кількість особистої інформації про своїх громадян і повинні гарантувати, що вона може бути використана тільки в дозволених цілях. Уряди держав також накопичують величезні обсяги інформації, пов'язаної з національною безпекою, яка вимагає застосування розгорнутих протоколів безпеки.

Крім очевидних заходів фізичної безпеки, пов'язаних зі зберіганням конфіденційної інформації в надійно захищеному приміщенні з контролем доступу та ефективною системою сигналізації, можна вжити низку заходів для захисту системи від широкого спектра атак.

Розглянемо смартфон. Технічно обізнана людина може розібрати корпус телефону і отримати доступ до апаратних засобів на рівні електронних схем.

Якщо ця людина зможе контролювати зовнішні електричні сигнали процесора і його канали зв'язку з іншими компонентами системи, які види інформації вона зможе зібрати? Відповідь залежить від типів і кількості апаратних засобів захисту, реалізованих у конструкції системи.

Першим кроком у проектуванні безпечної системи є запобігання внесенню вразливостей під час розроблення. При розробці системи, що містить вбудований процесор, дуже корисно передбачити інтерфейс **апаратного відладчика**, який дозволяє підключити ПК до пристрою за допомогою спеціального інтерфейсного кабелю. Використовуючи це з'єднання, розробники можуть перепрограмувати флеш-пам'ять, встановлювати контрольні точки в коді, відображати значення регістрів і змінних, а також покроково переглядати роботу коду. Якщо у випущеній версії проекту інтерфейс для підключення відладчика залишається на друкованій платі, то користувачі можуть підключити до системи власний відладчик і працювати з ним так само, як і розробники.

Це вочевидь небажано для будь-якої системи, призначеної для безпечної роботи. Оскільки можливість підключення відладчика залишається дуже корисною навіть після випуску продукту, розробники іноді намагаються залишити сигнали відладчика в схемі, але замаскувати їх якимось чином, щоб зробити їх функції менш очевидними. Певною мірою цей підхід може бути ефективним, однак завзяті хакери продемонстрували здатність виявляти наявність навіть наймайстерніше замаскованих налагоджувальних інтерфейсів і використовувати їх для доступу до внутрішніх пристроїв процесора. Наявність доступного інтерфейсу апаратного налагодження у випущеному продукті є серйозною вразливістю безпеки.

Багато виробників процесорів почали впроваджувати засоби захисту, щоб не дозволити навіть технічно обізнаним і цілеспрямованим зловмисникам порушити захист системи. Для забезпечення ефективного захисту розробники повинні впроваджувати і повною мірою використовувати ці можливості в конструкціях своїх систем. Ось деякі приклади технологій безпеки.

- **Захищений паролем інтерфейс апаратного відладчика.** Деякі сімейства процесорів підтримують додавання пароля до стандартного інтерфейсу апаратного налагодження. Перш ніж такий процесор увімкне функцію налагодження, під'єднана система зобов'язана пройти процес початкової автентифікації, під час якого вона має надати надійний пароль (наприклад, 256-

бітове число). Це ефективний підхід, який зберігає можливість безпечного пошуку та усунення проблем, що виникають після випуску продукту.

- **Внутрішній механізм шифрування зі сховищем ключів.** Деякі процесори підтримують можливість шифрування і дешифрування і зберігають секретні ключі для використання під час роботи. Ці ключі мають бути збережені в процесорі під час виготовлення системи і після збереження мають бути недоступні ззовні. Таке поєднання шифрувального механізму та збережених ключів забезпечує безпечний зв'язок з авторизованими зовнішніми пристроями. Використання високошвидкісного апаратного шифрування і дешифрування дозволяє забезпечити безпечний зв'язок з необхідною швидкістю між фізично розділеними підсистемами, наприклад в автомобілі.
- **Захист пам'яті пристрою.** Багато сімейств процесорів надають кілька варіантів захисту областей пам'яті. Наприклад, банк ПЗУ, що містить код, можна заблокувати після програмування, щоб гарантувати неможливість його подальшого перепрограмування. Області коду також можна заблокувати для читання у вигляді даних, але при цьому доступ до них для виконання може бути збережений. Процесори, у яких відсутній повноцінний блок керування пам'яттю, часто оснащують підсистемою, званою блоком захисту пам'яті (memory protection unit, MPU) і призначеною для керування вимогами безпеки різних областей пам'яті процесора.

Кілька можливостей, необхідних для забезпечення безпечної роботи системи, були об'єднані в пристрої під назвою Trusted Platform Module (довірений платформний модуль), про який піде мова в наступному розділі.

Довірений платформний модуль

Сучасні ПК містять підсистему, звану **довіреним платформним модулем** (Trusted Platform Module, TPM), який реалізує різні функції безпеки. TPM – це стійкий до злому процесор, розроблений спеціально для виконання криптографічних операцій. У сучасному ПК він може являти собою невеликий модуль, що приєднується до материнської плати, а іноді його реалізовано у вигляді окремого функціонального блоку в інтегральній схемі головного процесора.

Загальні категорії функціональних можливостей TPM такі.

- **Генерування випадкових чисел.** Коли необхідно вибрати велике число, яке буде дуже важко вгадати комусь іншому, найкраще використовувати для вибору цього числа дійсно випадковий метод. Більшість мов програмування надають механізм для генерування послідовностей чисел, що виглядають як випадкові, але в більшості випадків ці послідовності хоча б деякою мірою передбачувані після отримання серії вибірок. TPM надає функцію генерування істинних випадкових чисел, яка підходить для формування криптографічних ключів,

здатних забезпечити надійний захист даних. Генератор істинних випадкових чисел використовує спеціалізовані апаратні засоби для генерування непередбачуваної послідовності бітів 0 і 1.

- **Генерування криптографічних ключів.** Модуль TPM може створювати криптографічні ключі стандартних форматів для використання всередині системи та за запитом призначених для користувача додатків. Ці ключі зазвичай застосовуються для таких цілей, як цифровий підпис електронної пошти та захист доступу до вебдодатків.

- **Зберігання криптографічних ключів.** Крім генерування криптографічних ключів, TPM може надійно зберігати кілька ключів для різних цілей. Шифрування з відкритим ключем використовує попарно згенеровані ключі. Одним із ключів пари є відкритий ключ, яким можна вільно ділитися з ким завгодно. Інший ключ, званий секретним ключем, має бути захищений і призначений для використання тільки власником ключа. Дані зашифровані за допомогою відкритого ключа, можуть бути розшифровані лише за допомогою секретного ключа. І аналогічно, дані зашифровані за допомогою секретного ключа, можуть бути розшифровані тільки за допомогою відповідного відкритого ключа. Шифрування з відкритим ключем реалізує багато можливостей забезпечення безпеки, такі як шифрування електронної пошти та цифрові підписи документів. Модуль TPM можна налаштувати на постійне блокування секретного ключа, що зберігається в ньому, що гарантує неможливість його виявлення шкідливим програмним забезпеченням або навіть просунутим зловмисником, який отримав повний фізичний доступ до системи.

- **Перевірка цілісності системи.** TPM бере безпосередню участь у процесі безпечного завантаження. Цей процес гарантує, що під час завантаження дозволено виконання тільки довірених компонентів вбудованого та іншого програмного забезпечення. Процес безпечного завантаження містить обчислення і перевірку цифрового підпису кожного використовуваного під час завантаження компонента вбудованого та іншого програмного забезпечення до його виконання.

- **Моніторинг працездатності системи.** Модуль TPM може гарантувати, що функції безпеки системи, такі як шифрування дисків і безпечне завантаження, будуть безперервно діяти під час роботи системи.

- **Послуги аутентифікації.** Модуль TPM надає програмний інтерфейс, що пропонує криптографічні функції в міру необхідності для використання додатками непривілейованого рівня. Цей інтерфейс являє собою безпечний механізм прив’язки ідентифікаційних даних користувача до цифрового пристрою. Секретний ключ, що міститься в TPM, забезпечує надійну ідентифікацію користувача, що виключає можливість підробки.

Завдяки TPM сучасні ПК та інші цифрові пристрої можна безпечно використовувати у фінансових і конфіденційних транзакціях незахищеними каналами зв'язку, зокрема в Інтернеті.

Головна мета таких механізмів безпеки, як TPM, – запобігти спробам кібератак, метою яких є крадіжка інформації або фінансових активів, що належать користувачам комп'ютерних систем. Спектр можливих атак і деякі способи захисту від них обговорюються в наступному розділі.

Протидія кібератакам

Недостатньо спроектувати комп'ютерну систему зі стандартним набором засобів захисту і вважати, що хакери не зможуть подолати ці засоби захисту. Зловмисники демонструють надзвичайну кмітливість і використовують будь-які доступні способи, щоб проникнути у внутрішні механізми системи, яка здається їм цікавою, чи то з інтелектуальної цікавості, чи то зі злочинних мотивів.

Ось деякі області, які слід розглянути крім стандартних методів забезпечення безпеки.

- **Аналіз енергоспоживання, часових залежностей і випромінювань.** Використовуючи детальний моніторинг енергоспоживання системи, часу активності зовнішніх сигналів або навіть радіочастотних випромінювань, що генеруються під час виконання алгоритмів, можна провести зворотне розроблення дій, що відбуваються всередині процесора. Відомі випадки успішних атак, заснованих на цих методах, наприклад для вилучення секретних ключів шифрування під час виконання криптографічного алгоритму.
- **Порушення енергопостачання.** Атаки, що спричиняють коливання або падіння напруги живлення під час увімкнення або роботи системи, як було показано на практиці, призводять до того, що багато систем опиняються в уразливому стані, в якому деякі засоби захисту стають неефективними. Надійна система повинна передбачати таку поведінку, природну або навмисну, і повертатися в безпечний і надійний стан щоразу, коли джерело живлення працює поза межами встановлених параметрів.
- **Фізичне втручання.** Зловмисник може замінити деякі компоненти в системі, намагаючись отримати підвищений рівень контролю або доступ до важливих функцій чи даних. Наприклад, заміна завантажувального ПЗП у мобільному пристрої може забезпечити нормальну роботу системи й одночасно надати зловмиснику можливість отримати необмежений доступ до системи, використовуючи код у заміненому ПЗП. Дедалі більше сімейств процесорів підтримують використання цифрових підписів для перевірки автентичності коду ПЗП. Для перевірки вмісту ПЗП під час запуску процесор виконує криптографічний алгоритм хешування над вмістом ПЗП. Перш ніж буде дозволено виконання коду, що зберігається в ПЗП, результати обчислення хеш-

функції (сигнатура) повинні збігтися з сигнатурою, попередньо завантаженою у внутрішню пам'ять процесора. Алгоритм хешування розроблено таким чином, щоб практично унеможливити зміну даних, що зберігаються в ПЗП, і при цьому надати очікувану сигнатуру. Для додаткової безпеки вміст ПЗП також можна зашифрувати, що не дасть змоги зловмиснику проаналізувати код, який міститься в ньому.

У цьому розділі було описано сучасні підходи, які широко використовуються для реалізації заходів безпеки в цифрових пристроях, і розглянуто деякі з більш рідкісних вразливостей, якими користуються цілеспрямовані зловмисники. Проєктування безпечної системи має починатися з твердого засвоєння основних елементів безпеки, але також вимагає великої винахідливості при розгляді безлічі способів, якими рішуче налаштований зловмисник може спробувати прорвати її захист.

Резюме

У цьому розділі на підставі викладеного в попередніх розділах було представлено особливості комп'ютерної архітектури, що реалізують функціональні вимоги, характерні для конкретних галузей. Ми зосередилися на розширеннях, які зазвичай реалізують на рівні набору інструкцій процесора й у зовнішніх щодо процесора компонентах, які надають додаткові можливості крім загальних вимог до обчислень.

Тепер ви маєте добре орієнтуватися в привілейованих режимах процесора та їхньому використанні в багатопроцесорних і багатокористувацьких контекстах, концепціях процесорів і наборів інструкцій для обчислень із плаваючою комою, методах керування живленням у пристроях з акумуляторним живленням, а також в особливостях процесорів і систем, призначених для підвищення рівня їхнього захисту.

Ці знання підготували нас до наступного розділу, в якому ми розглянемо найбільш популярні архітектури процесорів і набори інструкцій, що використовуються в даний час в персональних комп'ютерах, комп'ютерах бізнес-класу та інтелектуальних мобільних пристроях. Будуть розглянуті такі архітектури: *x*86, *x*64, а також 32- і 64-розрядні варіанти ARM.

Вправи

1. Використовуючи мову програмування, яка надає доступ до байтового представлення типів даних з плаваючою комою (наприклад, C або C++), напишіть функцію, яка приймає як вхідні дані 32-бітове значення одинарної точності. Витягніть знак, порядок і мантису з байтів цього значення з плаваючою комою і виведіть їх на пристрій відображення. Видаліть зміщення з порядку перед виведенням його значення і виведіть мантису у вигляді десяткового числа. Протестуйте програму на значеннях `0`, `-0`, `1`, `-1`, `6.674e-11`, `1.0e38`, `1.0e39`, `1.0e-38` та `1.0e-39`. Перераховані тут

числові значення, що містять `e`, використовують текстове представлення чисел із плаваючою комою мовою C/C++. Наприклад, `6.674e-11` означає $6{,}674 * 10^{-11}$.

2. Змініть програму із *вправи 1*, щоб вона могла також приймати значення з плаваючою комою подвійної точності та виводити знак, порядок (з видаленим зміщенням) і мантису цього значення. Протестуйте її з тими самими вхідними значеннями, що й у *вправі 1*, а також зі значеннями `1.0e308, 1.0e309, 1.0e-308` і `1.0e-309`.
3. Знайдіть в Інтернеті інформацію про сімейство процесорів i.MX RT1060 компанії NXP Semiconductors. Скачайте технічні описи цього сімейства і дайте відповіді на такі запитання про ці процесори.
4. Чи підтримують процесори i.MX RT1060 концепцію виконання інструкцій у режимі супервізора? Поясніть свою відповідь.
5. Чи підтримують процесори i.MX RT1060 концепцію віртуальної пам’яті зі сторінковою організацією? Поясніть свою відповідь.
6. Чи підтримують процесори i.MX RT1060 апаратні обчислення з плаваючою комою? Поясніть свою відповідь.
7. Які функції керування живленням підтримують процесори i.MX RT1060?
8. Які функції безпеки підтримують процесори i.MX RT1060?

10

Сучасні архітектури і набори інструкцій процесорів

Більшість сучасних персональних комп'ютерів містять процесор, що підтримує 32-розрядну архітектуру *x*86 або 64-розрядну архітектуру *x*64 від Intel або AMD. З іншого боку, майже всі смартфони, розумні годинники, планшети та багато вбудованих систем містять 32- або 64- розрядні процесори ARM. У цьому розділі докладно розглядаються регістри і набори інструкцій процесорів цих сімейств.

Прочитавши цей розділ, ви отримаєте уявлення про високорівневу організацію та унікальні атрибути регістрів, наборів інструкцій і мов асемблера архітектур *x*86, *x*64, 32- і 64-розрядних архітектур ARM, а також про важливі аспекти успадкованих функцій, які підтримуються в цих архітектурах.

У цьому розділі розглядаються такі теми:

- архітектура та набір інструкцій *x*86;
- архітектура та набір інструкцій *x*64;
- архітектура і набір інструкцій 32-розрядних процесорів ARM;
- архітектура та набір інструкцій 64-розрядних процесорів ARM.

Технічні вимоги

Файли для цього розділу, включно з відповідями до вправ, доступні за адресою **https://github.com/PacktPublishing/Modern-Computer-Architecture-and-OrgяnizaйonSecond-Edition.**

Архітектура та набір інструкцій x86

Для цілей цього обговорення термін **"*x*86"** відноситься до 16-розрядної та 32-розрядної версій архітектури набору інструкцій для серії процесорів, що розпочалася з Intel 8086, представленого в 1978 р. Процесор 8088, випущений 1979 р., функціонально дуже схожий на 8086, за винятком того, що він оснащений 8-розрядною шиною даних замість 16-розрядної шини в 8086. 8088 був центральним процесором оригінального персонального комп'ютера IBM PC.

Наступні покоління процесорів цієї серії отримали назви 80186, 80286, 80386 і 80486, що призвело до появи терміну "x86" як скорочення для всіх членів цього сімейства. Надалі було ухвалено рішення про відмову від угоди про цифрові найменування, і для наступних поколінь використовували назви Pentium, Core, i Series, Celeron і Xeon.

Advanced Micro Devices (AMD), компанія з виробництва напівпровідників, що конкурує з Intel, виробляє x86-сумісні процесори з 1982 р. Деякі з недавніх поколінь процесорів AMD *x*86 отримали назви Ryzen, Opteron, Athlon, Turion, Phenom і Sempron.

Сумісність щодо виконання коду між процесорами Intel і AMD є гарною в багатьох аспектах. Проте між процесорами цих двох виробників є деякі суттєві відмінності, зокрема конфігурація виводів чіпів і сумісність із набором мікросхем.

Процесори Intel, як правило, працюють тільки на материнських платах і з наборами мікросхем, призначеними для чіпів Intel, а процесори AMD – тільки на материнських платах і з наборами мікросхем, призначеними для чіпів AMD. Пізніше в цьому розділі ми розглянемо деякі інші відмінності між процесорами Intel і AMD.

8086 і 8088 – це 16-розрядні процесори, незважаючи на 8-розрядну шину даних у 8088. Внутрішні регістри в цих процесорах мають розрядність 16 біт, а набір інструкцій працює з 16-бітними значеннями даних. Процесор 8088 явно виконує два цикли шини для передачі кожного 16-бітного значення між процесором і пам'яттю.

8086 і 8088 не підтримують складніші функції сучасних процесорів, як-от віртуальна пам'ять зі сторінковою організацією та кільця захисту. Ці ранні процесори також мали всього 20 адресних рядків, що обмежувало обсяг адресованої пам'яті одним мегабайтом. 20-розрядна адреса не може поміститися в 16-розрядний регістр, тому для доступу до повного адресного простору об'ємом 1 Мбайт необхідно використовувати деяким чином ускладнену систему сегментних регістрів і зміщень.

У 1985 р. компанія Intel випустила процесор 80386 з поліпшеннями, які зняли багато з цих обмежень.

У ньому було запроваджено такі можливості.

- **32-розрядна архітектура.** Адреси, регістри та АЛУ мають розрядність 32 біти, а інструкції спочатку працюють з операндами розрядністю до 32 біт.

- **Захищений режим.** Цей режим реалізує багаторівневий механізм привілеїв, що складається з кілець із номерами від 0 до 3, які ми розглянули в *розділі 9*. У Windows і Linux кільце 0 – це режим ядра, а кільце 3 – користувацький режим. Кільця 1 і 2 не використовуються в цих операційних системах.
- **Блок управління пам'яттю (MMU) на чипі.** У процесорі 80386 блок MMU підтримує плоску модель пам'яті, що дає змогу за допомогою 32-розрядної адреси одержати доступ до будь-якого осередку в просторі об'ємом 4 Гбайт. Маніпулювання сегментними регістрами і зсувами більше не потрібне. MMU підтримує віртуальну пам'ять зі сторінковою організацією.
- **Триступеневий конвеєр інструкцій.** Цей конвеєр прискорює виконання інструкцій, як описано в *розділі 8*.
- **Регістри апаратного налагодження.** Регістри налагодження підтримують налаштування до чотирьох контрольних точок, які зупиняють виконання коду за вказаною віртуальною адресою під час звернення до цієї адреси та виконання обраної умови. Доступні умови зупинки – виконання коду, запис даних і читання або запис даних. Ці регістри доступні тільки для використання кодом, запущеним у кільці 0.

Сучасні процесори сімейства 386 під час завантаження переводяться в 16-розрядний режим роботи оригінального 8086, який тепер називається **реальним режимом**. Він зберігає сумісність із програмним забезпеченням, написаним для середовища 8086/8088, таким як операційна система MS-DOS.

У більшості сучасних систем, що працюють на процесорах сімейства *x*86, перехід у захищений режим відбувається під час запуску системи. Після цього переходу операційна система залишається в цьому режимі доти, доки комп'ютер не буде вимкнено.

MS-DOS НА СУЧАСНОМУ ПК



Процесор сімейства *x*86 у сучасному ПК сумісний на рівні інструкції з оригінальним процесором 8088, однак запуск старої копії MS-DOS у сучасній комп'ютерній системі навряд чи буде простою процедурою. Периферійні пристрої та їхні інтерфейси в сучасних ПК не сумісні з відповідними інтерфейсами в ПК 1980-х років. MS-DOS знадобиться драйвер, який розуміє, як взаємодіяти, наприклад, з підключеною через інтерфейс USB клавіатурою сучасної материнської плати.

У наші дні 16-розрядний режим у процесорах *x*86 здебільшого використовується як завантажувач для операційної системи, що працює в захищеному режимі. Оскільки більшість розробників комп'ютерних пристроїв і програмного забезпечення, що працює на них, навряд чи будуть залучені до реалізації такої можливості, решта

нашого обговорення архітектури *x*86 у цьому розділі буде присвячена захищеному режиму і пов'язаній з ним плоскій моделі 32-розрядної пам'яті.

Архітектура *x*86 підтримує беззнакові та знакові цілочисельні типи даних у додатковому коді розрядністю 8, 16, 32, 64 і 128 біт. Цим типам даних присвоєно такі назви:

- **byte** (байт) – 8 біт;
- **word** (слово) – 16 бітів;
- **doubleword** (подвійне слово) – 32 біти;
- **quadword** (чотирикратне слово) – 64 біти;
- **double quadword** (подвійне чотирикратне слово) – 128 біт.

У більшості випадків архітектура *x*86 не наказує зберігання даних цих типів у природних межах. **Природна границя** типу даних – це будь-яка адреса, що ділиться на розмір типу даних у байтах без залишку.

Зберігання будь-якого з багатобайтових типів у невирівняних межах дозволено, але не рекомендується, оскільки це негативно позначається на продуктивності: для виконання Інструкцій, які працюють із невирівняними даними, потрібні додаткові такти. Кілька інструкцій, які працюють із подвійними чотирикратними словами, вимагають сховища з природним вирівнюванням і під час спроби доступу до невирівняних даних генерують помилку загального захисту.

Архітектура *x*86 має вбудовану підтримку типів даних із плаваючою комою з розрядністю 16, 32, 64 і 80 бітів. 32-бітний, 64-бітний і 80-бітний формати були представлені в *розділі 9*. 16-бітний формат із плаваючою комою називають форматом **половинної точності**, він має 11-бітну мантису, мається на увазі провідний біт 1 і 5-бітний порядок. Формат із плаваючою комою половинної точності широко використовується в обробці даних графічних процесорів.

У наступному розділі ми детально розглянемо набір регістрів архітектури *x*86.

Набір регістрів архітектури x86

У захищеному режимі архітектури *x*86 передбачено вісім 32-розрядних регістрів загального призначення, регістр прапорів і покажчик інструкцій. Також передбачено шість сегментних регістрів і додаткові регістри конфігурації, що залежать від конкретної моделі процесора. Сегментні регістри і регістри, які стосуються конкретної моделі процесора, налаштовуються системним програмним забезпеченням під час запуску і як правило, не становлять ніякого інтересу для розробників додатків і драйверів пристроїв. Із цих причин ми не будемо далі обговорювати сегментні регістри та регістри для конкретних моделей процесорів.

16-розрядні регістри загального призначення в оригінальній архітектурі 8086 мають назви AX, CX, DX, BX, SP, BP, SI та DI. Причина перерахування перших чотирьох регістрів у порядку, що не відповідає алфавітному, полягає в тому, що ці вісім регістрів поміщаються в стек за допомогою інструкції `pushad` (помістити всі регістри) саме в такому порядку.

При переході на 32-розрядну архітектуру процесора 80386 розрядність кожного регістра збільшилася до 32 біт. 32-розрядна версія імені регістра має префікс із буквою E для позначення цього розширення.

Можливий доступ до частин 32-розрядних регістрів з меншою бітовою шириною. Наприклад, до молодших 16 бітів 32-розрядного регістра EAX можна звертатися за ім'ям AX. Регістр AX, своєю чергою, може бути доступний у вигляді окремих байтів, використовуючи імена AH (старший байт) і AL (молодший байт). На рис. 10.1 показано імена регістрів і підмножини кожного з них.

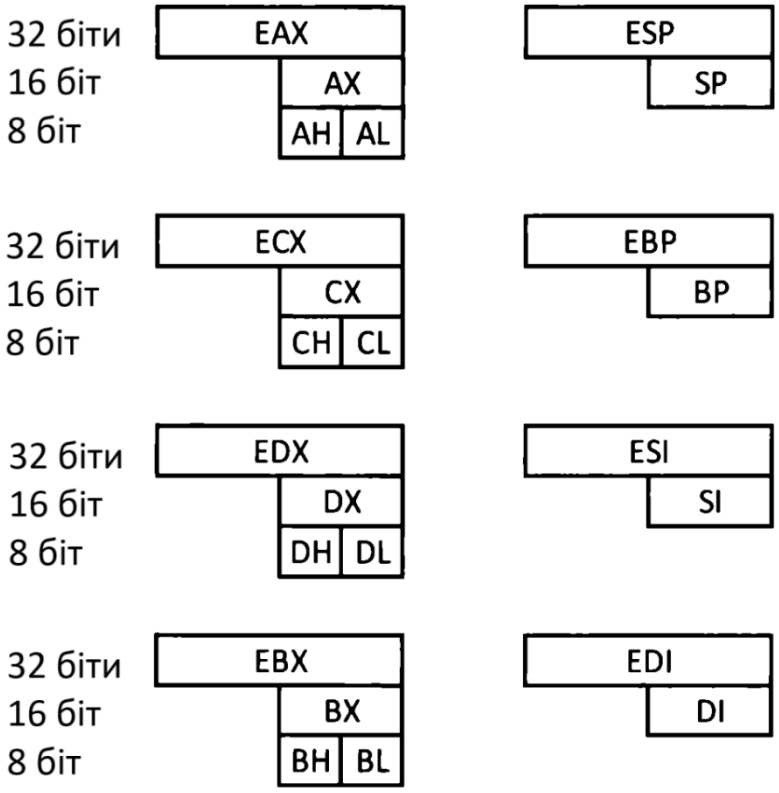


Рис 10.1. *Імена та підмножини регістрів*

Запис у частину 32-розрядного регістра, наприклад в AL, зачіпає тільки біти в цій частині. Завантаження 8-бітного значення в AL змінює молодші 8 біт регістра EAX, залишаючи інші 24 біти без змін.

У порядку дотримання традиції архітектури CISC сімейства *x*86 кілька функцій, пов'язаних із різними інструкціями, прив'язані до певних регістрів. У табл. 10.1 наведено опис функцій, пов'язаних із кожним регістром загального призначення x86.

Таблиця 10.1. *Регістри загального призначення x86 і пов'язані з ними функції*

Регістр	Назва	Функція
EAX	Акумулятор	Арифметичні операції
ECX	Лічильник	Лічильник циклів і лічильник зсуву/обертання
EDX	Дані	Арифметичні операції та операції введення-виведення
EBX	База	Покажчик на дані
ESP	Покажчик стека	Вказівник на вершину стека
EBP	Покажчик бази	Покажчик на базу стека всередині функції
ESI	Індекс джерела	Покажчик на вихідне місце розташування в операціях з масивами
EDI	Індекс приймача	Покажчик на кінцеве місце розташування в операціях з масивами

Ці прив'язані до регістрів функції контрастують з архітектурами багатьох процесорів RISC, які, як правило, пропонують більшу кількість регістрів загального призначення. Регістри в RISC-процесорі здебільшого функціонально еквівалентні один одному.

Регістр прапорців *x*86, `EFLAGS`, містить біти стану процесора, описані в табл. 10.2.

Таблиця 10.1. *Регістри загального призначення x86 і пов'язані з ними функції*

Біт	Назва	Функція
0	CF	Прапор перенесення: вказує, чи призвело додавання до перенесення або віднімання до запозичення. Використовується як вхідні дані інструкціями додавання та віднімання
2	PF	Прапор парності: встановлюється, якщо молодші 8 біт результату містять парне число бітів зі значенням 1
4	AF	Прапор допоміжного перенесення: вказує, чи призвело додавання до перенесення або віднімання до запозичення з 4 молодших бітів. Використовується у двійково-десятковій арифметиці
6	ZF	Прапор нуля: встановлюється, якщо результат операції дорівнює нулю
7	SF	Прапор знаку: встановлюється, якщо результат операції негативний
8	TF	Прапор трасування: використовується під час покрокового налагодження
9	IF	Прапор дозволу переривань: встановлення цього біта дозволяє виконання апаратних переривань
10	DF	Прапор напрямку: керує напрямком обробки рядків. Якщо дорівнює 0, використовується порядок від нижчої адреси до вищої. Якщо дорівнює 1, використовується порядок від найвищої адреси до найнижчої

Таблиця 10.1. *(закінчення)*

Біт	Назва	Функція
11	OF	Прапор переповнення: встановлюється, якщо операція призвела до переповнення зі знаком
12–13	IOPL	Рівень привілеїв введення-виведення: рівень привілеїв потоку, що виконується в цей момент. Якщо IOPL має значення 0 – це режим ядра, якщо дорівнює 3 – користувацький режим
14	NT	Прапор вкладеного завдання: керує ланцюжком переривань
16	RF	Прапор відновлення: використовується для обробки винятків під час налагодження
17	VM	Прапор віртуального режиму 8086: якщо встановлено, увімкнено режим сумісності з процесором 8086. Цей режим дає змогу запускати деякі додатки MS-DOS у контексті операційної системи із захищеним режимом
18	AC	Прапор контролю вирівнювання: якщо встановлено, увімкнено функцію контролю вирівнювання даних у пам’яті. Наприклад, якщо прапор AC встановлено, збереження 16-бітного значення за непарною адресою спричиняє виняток контролю вирівнювання. Коли цей прапор не встановлено, процесори x86 можуть виконувати доступ до пам’яті без вирівнювання, але кількість тактів, необхідних для виконання інструкцій, при цьому може збільшитися.
19	VIF	Прапор дозволу віртуальних переривань: віртуальна версія прапора IF у віртуальному режимі 8086
20	VIP	Прапор віртуального переривання, що очікує: встановлюється, коли переривання перебуває в стані очікування у віртуальному режимі 8086
21	ID	Прапор ідентифікації: якщо цей біт може бути встановлений, підтримується інструкція `cpuid`. Інструкція `cpuid` повертає ідентифікатор процесора та інформацію про функції

Усі біти в регістрі `EFLAGS`, яких немає в табл. 10.2, зарезервовані й не використовуються.

32-бітний покажчик інструкцій `EIP` містить адресу наступної інструкції для виконання, якщо не було вибрано відгалуження. Коли виконується умова від-відгалуження, в `EIP` завантажується адреса призначення гілки і виконання продовжується з цієї адреси.

В архітектурі x86 прийнято прямий порядок байтів. Це означає, що багатобайтові значення зберігаються в пам’яті з найменшим значущим байтом за найнижчою адресою і найзначущішим байтом за вищою адресою.

Режими адресації x86

Як і слід було очікувати від архітектури CISC, *x*86 підтримує кілька режимів адресації. Існує кілька правил, пов'язаних з адресацією операндів-джерел і операндів-приймачів, яких необхідно дотримуватися для отримання правильних інструкцій. Наприклад, розміри операнда-джерела й операнда-приймача в інструкції `mov` мають бути рівними. Асемблер спробує вибрати відповідний розмір для операнда, який має невизначений розмір (наприклад, безпосереднє значення 7), щоб він відповідав розрядності цільового місця розташування (наприклад, 32-бітного регістра `EAX`). У випадках, коли розмір операнда не може бути визначений, для зазначення розміру необхідно використовувати ключові слова, такі як `byteptr`.

Мова асемблера в цих прикладах використовує *синтаксис Intel*, у якому прийнято розміщення операндів у порядку "приймач – джерело". Синтаксис Intel використовується в основному в контекстах Windows і MS-DOS. В альтернативній нотації, відомій як синтаксис *AT&T*, операнди розміщуються в порядку "джерело – приймач". Цей синтаксис використовується в операційних системах на основі UNIX. У всіх прикладах у цій книзі використовується синтаксис Intel.

Архітектура *x*86 підтримує кілька режимів адресації, які ми розглянемо далі. Коментарі в коді асемблера починаються з крапки з комою і тривають до кінця рядка.

Неявна адресація

У цьому режимі адресації регістр визначається опкодом інструкції. Приклад:

```
clc ; скидання прапора перенесення (CF в регістрі прапорів EFLAGS)
```

Регістрова адресація

Один або обидва регістри, джерело і приймач, закодовані в інструкції:

```
mov eax, ecx ; копіювання вмісту регістра ECX в регістр EAX
```

Регістри можна використовувати як перший операнд, другий операнд або обидва операнди.

Безпосередня адресація

Як операнд інструкції підставляється безпосереднє значення:

```
mov eax, 7 ; переміщення 32-бітного значення 7 в регістр EAX
mov ax, 7 ; переміщення 16-бітного значення 7 в регістр AX
```

При використанні синтаксису Intel не потрібно ставити перед безпосередніми значеннями символ #.

Адресації з прямим доступом до пам'яті

Як операнд інструкції підставляється адреса значення:

```
mov eax, [078bch] ; копіювання 32-бітного значення за шістнадцятковою адресою
                  ; 78BC в регістр EAX
```

В асемблері *x*86 поміщення виразу в квадратні дужки означає, що вираз є адресою. При виконанні переміщень або інших операцій над операндами, укладеними в квадратні дужки, значення, над яким виконується операція, – це дані за вказаною адресою. Винятком із цього правила є інструкція LEA (завантаження дійсної адреси), яку ми розглянемо пізніше.

Непряма регістрова адресація

Операндом є регістр, що містить адресу значення даних:

```
mov eax, [esi] ; копіювання 32-бітного значення за адресою, вказаною
               ; в регістрі ESI, в регістр EAX
```

Цей режим еквівалентний використанню покажчика для посилання на змінну в C або C++.

Індексна адресація

Операнд вказує регістр і зміщення, поєднання яких визначає адресу значення даних:

```
mov eax, [esi + 0bh] ; копіювання 32-бітного значення за адресою (ESI + 0bh)
                     ; в регістр EAX
```

Цей режим зручний для доступу до елементів структури даних. У цьому сценарії регістр ESI містить адресу структури, а додана константа – байтове зміщення елемента від початку цієї структури.

Відносна індексна адресація

Операнд вказує базовий регістр, індексний регістр і зміщення, які підсумовуються для визначення адреси значення даних:

```
mov eax, [ebx + esi + 10] ; копіювання 32-бітного значення з початком
                          ; за адресою (EBX + ESI + 10) в регістр EAX
```

Режим зручний для доступу до окремих елементів даних у масиві структур даних. У цьому прикладі регістр EBX містить адресу початку масиву структур, регістр ESI містить зсув потрібної структури в масиві, а постійне значення (10) являє собою зсув потрібного елемента даних від початку обраної структури.

Відносна індексна адресація з масштабуванням

Операнд складається з базового регістра, індексного регістра, помноженого на масштабний коефіцієнт, і зміщення, які підсумовуються для визначення адреси значення даних:

```
mov eax, [ebx + esi*4 + 10] ; копіювання 32-бітного значення з початком
                            ; за адресою (EBX + ESI*4 + 10) в регістр EAX
```

У цьому режимі адресації перед підсумовуванням з іншими компонентами адреси операнда значення в індексному регістрі може бути помножено на 1 (за замовчуванням), 2, 4 або 8. Використання масштабного коефіцієнта не призводить до зниження продуктивності. Ця можливість зручна під час ітерації по масивах, що містять елементи розміром 2, 4 або 8 байт.

У режимах відносної адресації як базовий або індексний регістр можна використовувати більшість регістрів загального призначення.

На рис. 10.2 показано можливі комбінації використання регістрів і масштабування в режимах відносної адресації.

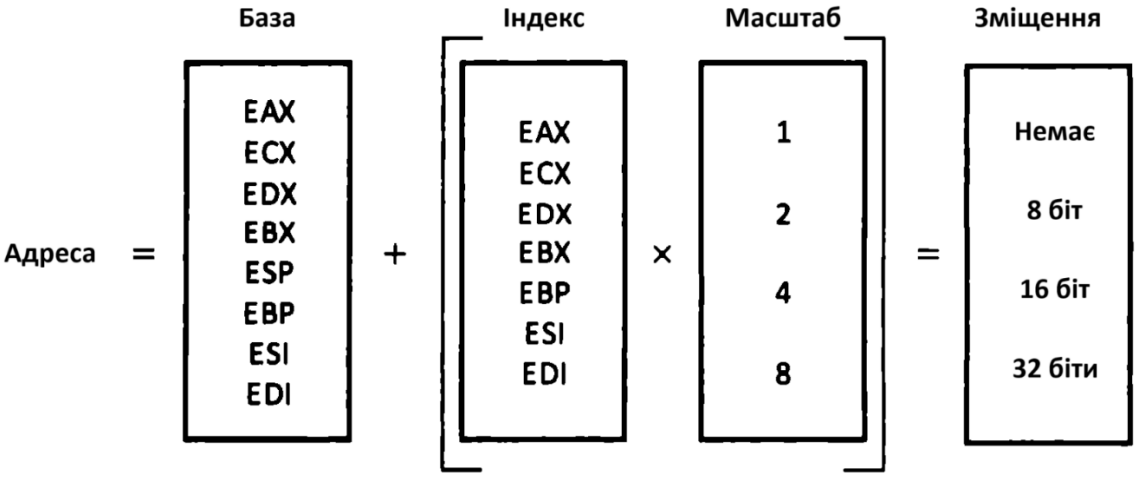


Рис 10.2. *Режим відносної адресації*

Усі вісім регістрів загального призначення можна використовувати як базовий регістр. І з цих восьми тільки `ESP` не можна застосовувати як індексний регістр.

Категорії інструкцій x86

Набір інструкцій x86 був представлений разом із процесором Intel 8086 і за минулі роки кілька разів розширювався. Деякі з найзначніших змін пов'язані з розширенням архітектури з 16 до 32 біт, під час якого також було додано захищений режим і віртуальну пам'ять зі сторінковою організацією. Майже у всіх випадках нові можливості додавалися з збереженням повної зворотної сумісності.

Повний набір інструкцій *x*86 містить кілька сотень інструкцій. Ми не будемо обговорювати їх усі в цьому розділі. У цьому розділі наведені короткі огляди найбільш важливих інструкцій, які найчастіше трапляються, застосовних до додатків, які працюють у користувацькому режимі, і драйверів пристроїв.

Цю підмножину інструкцій *x*86 можна розділити на кілька загальних категорій: переміщення даних, маніпулювання стеком, арифметика і логіка, перетворення, потік керування, маніпулювання рядками і прапорами, введення-виведення, захищений режим. Ми також розглянемо кілька допоміжних інструкцій, які не належать до будь-якої певної категорії.

Переміщення даних

Інструкції переміщення даних не впливають на прапори процесора. Переміщення даних виконують такі інструкції.

- `mov` – копіювання значення даних, на яке вказує другий операнд, у комірку пам'яті, зазначену як перший операнд.
- `cmovcc` – умовне переміщення даних другого операнда в регістр, зазначений як перший операнд, якщо умова `cc` істинна. Умова визначається за одним або кількома з таких прапорів процесора: `CF`, `ZF`, `SF`, `OF` і `PF`. І використовуються такі коди умов: `e` (дорівнює), `ne` (не дорівнює), `g` (більше), `ge` (більше або дорівнює), `a` (вище), `ae` (вище або дорівнює), `l` (менше), `le` (менше або дорівнює), `b` (нижче), `be` (нижче або дорівнює), `o` (переповнення), `no` (без переповнення), `z` (нуль), `nz` (не нуль), `s` (`SF` = 1), `ns` (SF = 0), `cxz` (регістр `CX` містить нуль) і `ecxz` (регістр `ECX` містить нуль).
- `movsx`, `movzv` – це варіанти інструкції `mov`, що виконують розширення знаку і доповнення нулями, відповідно. Операнд-джерело має бути меншого розміру, ніж операнд-приймач.
- `lea` – обчислення адреси, що надається другим операндом, і збереження її в комірку пам'яті, зазначену в першому операнді. Другий операнд укладений у квадратні дужки. На відміну від інших інструкцій переміщення даних, за адресою призначення зберігається обчислена адреса, а не значення даних.

Маніпулювання стеком

Інструкції маніпулювання стеком не впливають на прапори процесора. До цієї категорії належать такі інструкції:

- `push` – зменшення вмісту регістра `ESP` на 4, а потім поміщення 32-бітного операнда в комірку стека, на яку вказує `ESP`.

- pop – копіювання 32-бітного значення даних, на яке вказує вміст регістра ESP, у місце розташування операнда (регістр або адреса в пам'яті), а потім збільшення вмісту ESP на 4.
- pushfd, popfd – поміщення в стек або витяг зі стека регістра EFLAGS.
- pushad, popad – поміщення в стек або витяг зі стека регістрів EAX, ECX, EDX, EBX, ESP, EBP, ESI та EDI у зазначеному порядку.

Арифметика і логіка

Арифметичні та логічні інструкції змінюють прапори процесора. Наступні інструкції виконують арифметичні та логічні операції.

- add, sub – цілочисельне додавання і віднімання. Під час віднімання другий операнд віднімається з першого. Обидва операнди можуть бути регістрами, або один операнд може бути коміркою пам’яті, а інший – регістром. Один операнд може бути константою.
- adc, sbb – цілочисельне додавання і віднімання з використанням прапора CF як вхідних даних – ознаки перенесення (для додавання) або запозичення (для віднімання).
- cmp – віднімання одного операнда з іншого з відкиданням результату й одночасним оновленням прапорів OF, SF, ZF, AF, PF і CF на підставі результату.
- neg – заперечення операнда.
- inc, dec – збільшення або зменшення операнда на одиницю.
- mul – множення беззнакових цілих чисел. Розмір добутку залежить від розміру операнда. Операнд розміром у байт множиться на AL, а результат поміщається в AX. Операнд розміром у слово множиться на AX, а результат поміщається в DX:AX зі старшими 16 бітами в DX. Операнд розміром у подвійне слово множиться на EAX, а результат поміщається в EDX:EAX.
- imul – множення знакових цілих чисел. Перший операнд має бути регістром, і в нього ж поміщається результат операції. Усього може бути два або три операнди. У формі з двома операндами перший операнд множиться на другий операнд, а результат зберігається в першому операнді (регістрі). У формі з трьома операндами другий операнд множиться на третій операнд, а результат зберігається в першому операнді (регістрі). У формі з трьома операндами третій операнд має бути безпосереднім значенням.
- div, idiv – беззнакове (div) або знакове (idiv) ділення. Розмір результату залежить від розміру операнда. Операнд розміром у байт ділиться на AX, частка

поміщається в AL, а залишок – у AH. Операнд розміром у слово ділиться на DX:AX, частка поміщається в AX, а залишок – у DX. Операнд розміром у подвійне слово ділиться на EDX:EAX, частка поміщається в EAX, а залишок – в EDX.

- and, or, xor – відповідна логічна операція над двома операндами та збереження результату в місці розташування операнда-приймача.
- not – логічна операція НІ (інверсія бітів) над одним операндом.
- sal, shl, sar, shr – логічний (shl і shr) або арифметичний (sal і sar) зсув аргументу розміром у байт, слово або подвійне слово ліворуч або праворуч на задану кількість позицій – від 1 до 31 біта. sal і shl поміщають останній зрушений біт у прапор перенесення та вставляють нулі в молодші значущі біти, які звільнилися. shr поміщає останній зсунутий біт у прапор перенесення і вставляє нулі у звільнені старші біти. sar відрізняється від shr поширенням біта знаку у звільнені старші біти.
- rol, rcl, ror, rcr – операція обертання вліво або вправо на 0–31 біт, на вибір – за допомогою прапора перенесення. rcl і rcr виконують операцію обертання з використанням прапора перенесення, а rol і ror - без його використання.
- bts, btr, btc – зчитування біта з заданим номером (представленим як другий операнд) з першого операнда у прапор перенесення, а потім встановлення (bts), скидання (btr) або доповнення (btc) цього біта. Цим інструкціям може передувати ключове слово lock, що робить цю операцію неподільною.
- test – логічна операція І над двома операндами та оновлення стану прапорів SF, ZF і PF залежно від результату.

Перетворення

Інструкції перетворення збільшують розмірність значень даних. До цієї категорії належать такі інструкції.

- cbw – перетворення байта (регістр AL) у слово (регістр ax).
- cwd – перетворення слова (регістр ax) у подвійне слово (регістр DX).
- cwde – перетворення слова (регістр AX) у подвійне слово (регістр EAX).
- cdq – перетворення подвійного слова (регістр ax) в зчетверене слово (регістр EDX:EAX).

Потік керування

Інструкції потоку керування здійснюють умовну або безумовну передачу виконання на певну адресу.

- `jmp` – передача управління інструкції, розташованої за адресою, яка вказана як операнд.
- `jcc` – передача керування інструкції, розташованої за адресою, яка вказана як операнд, якщо умова `cc` істинна. Коди умов були наведені раніше в описі інструкції `cmovcc`. Умова визначається за одним або кількома з таких прапорів процесора: `CF`, `ZF`, `SF`, `OF` і `PF`.
- `call` – поміщення поточного значення регістра `EIP` у стек і передача керування інструкції за адресою, вказаною як операнд.
- `ret` – витяг верхнього значення зі стека і збереження його в регістрі `EIP`. Якщо є операнд, ця інструкція витягує вказану кількість байтів зі стека для очищення параметрів.
- `loop` – зменшення на одиницю значення лічильника `loop` у регістрі `ECX` і, якщо воно не дорівнює нулю, передача керування інструкції, розташованої за адресою, що вказана як операнд.

Маніпулювання рядками

Інструкції маніпулювання рядками можуть мати префікс у вигляді ключового слова `rep` для повторення операції задану кількість разів, що вказується в регістрі `ECX`, зі збільшенням або зменшенням на одиницю місця розташування джерела і приймача на кожній ітерації, залежно від стану прапора `DF`. Розмір операнда, оброблюваного на кожній ітерації, може дорівнювати байту, слову або подвійному слову. Початкова адреса кожного рядкового елемента задається в регістрі `ESI`, а адреса призначення – в регістрі `EDI`. До цієї категорії належать наступні інструкції:

- `mov` – переміщення рядкового елемента.
- `cmps` – порівняння елементів у відповідних позиціях у межах двох рядків.
- `scas` – порівняння рядкового елемента зі значенням у регістрі `EAX`, `AX` або `AL`, залежно від розміру операнда.
- `lods` – завантаження рядка в регістр `EAX`, `AX` або `AL`, залежно від розміру операнда.
- `stos` – збереження вмісту регістра `EAX`, `AX` або `AL`, залежно від розміру операнда, за адресою, записаною в регістр `EDI`.

Маніпулювання прапорами

Інструкції маніпулювання прапорами змінюють біти в регістрі `EFLAGS`.

- `stc`, `clc`, `cmc` – встановлення, скидання або доповнення прапора перенесення `CF`.
- `std`, `cld` – встановлення або скидання прапора напрямку `DF`.
- `sti`, `cli` – встановлення або скидання прапора переривання `IF`.

Введення-виведення

Інструкції введення-виведення зчитують дані з периферійних пристроїв або записують у них дані.

- `in`, `out` – переміщення 1, 2 або 4 байтів між регістром `EAX`, `AX` або `AL` і портом введення-виведення залежно від розміру операнда.
- `ins`, `outs` – переміщення елемента даних між пам'яттю і портом введення-виведення так само, як це роблять інструкції для роботи з рядками.
- `rep ins`, `rep outs` – переміщення блоків даних між пам'яттю і портом введення-виведення так само, як це роблять інструкції для роботи з рядками.

Захищений режим

Наступні інструкції відкривають доступ до функцій захищеного режиму.

- `sysenter`, `sysexit` – передача керування з кільця 3 на кільце 0 (`sysenter`) або з кільця 0 на кільце 3 (`sysexit`) в процесорах Intel.
- `syscall`, `sysret` – передача керування з кільця 3 на кільце 0 (`syscall`) або з кільця 0 на кільце 3 (`sysret`) в процесорах AMD. У режимі x86 (32-розрядному) процесори AMD також підтримують інструкції `sysenter` і `sysexit`.

Допоміжні інструкції

Ці інструкції не підпадають під категорії, перелічені вище.

- `int` – ініціювання програмного переривання. Операнд – це номер вектора переривання.
- `nop` – без операції.
- `cpuid` – надання інформації про модель процесора та його можливості.

Інші категорії інструкцій

Інструкції, перелічені в цьому розділі, є одними з найпоширеніших у додатках *x*86 і драйверах пристроїв, крім описаних у попередніх розділах. Архітектура *x*86 містить широке розмаїття категорій інструкцій, включно з такими:

- **інструкції обчислень із плаваючою комою** – ці інструкції виконуються в блоках обчислень із плаваючою комою *x*86;
- **інструкції SIMD** – до цієї категорії відносяться інструкції MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2 і AVX-512; деякі з наборів інструкцій у цій категорії були представлені в розд. III глави 8. *"Модель обробки SIMD" розділу 8*;
- **інструкції AES** – ці інструкції підтримують шифрування і дешифрування з використанням **вдосконаленого стандарту шифрування** (advanced encryption standard, AES);
- **інструкції MPX: розширення для захисту пам'яті** (memory protection eXtensions, MPX) покращують цілісність пам'яті, запобігаючи помилкам, таким як переповнення буфера;
- **інструкції SMX: розширення безпечного режиму** (Safer mode eXtensions, SMX) підвищують безпеку системи за наявності рішень, пов'язаних із довірою користувачам;
- **інструкції TSX: розширення для синхронізації транзакцій** (transac-tional synchronization eXtensions, TSX) підвищують продуктивність під час багатопотокового виконання з використанням загальних ресурсів;
- **інструкції VMX: розширення для віртуальних машин** (virtual machine eXtensions, VMX) забезпечують безпечну та ефективну підтримку віртуалізованих операційних систем.

Для інструментарію з плаваючою комою і SIMD передбачено додаткові регістри процесора.

Крім перерахованих тут існує ще низка категорій інструкцій *x*86, частину яких було видалено в пізніших поколіннях архітектури.

Загальні шаблони використання інструкцій

Нижче наведені приклади шаблонів використання інструкцій, з якими ви будете часто стикатися в скомпільованому коді. Методи, показані в цих прикладах, дають бажаний результат, зводячи при цьому до мінімуму розмір коду і необхідну кількість тактів:

```
xor reg, reg ; скидання reg
test reg, reg ; перевірка reg на рівність нулю
add reg, reg ; зсув reg вліво на один біт
```

Формати інструкцій x86

Окремі інструкції *x*86 мають змінну довжину, яка може змінюватися від 1 до 15 байт. Компоненти однієї інструкції, включно з будь-якими необов'язковими байтами, розміщуються в пам'яті в переліченій далі послідовності.

- **Байти префікса**. Один або кілька необов’язкових байтів префікса надають допоміжну інформацію про виконання операції з даним опкодом. Наприклад, префікс `lock` блокує шину в багатопроцесорній системі для виконання неподільних операцій типу "перевірити – встановити". Префікс `rep` і його варіанти дають змогу строковим інструкціям виконувати повторювані операції над строковими елементами в межах однієї інструкції. Інші префікси надають вказівки для інструкцій умовного розгалуження або для перевизначення прийнятого за замовчуванням розміру адреси або операнда.
- **Байти коду операції (опкоду).** За будь-якими байтами префікса йде оп-код архітектури *x*86, що складається з 1-3 байтів. Для деяких кодів операцій додаткові три біти опкоду кодуються в байті *ModR/M*, що йде за цим опкодом.
- **Байт ModR/M.** Цей байт потрібен не для всіх інструкцій. Байт *ModR/M* містить три інформаційні поля, що надають інформацію про режим адресації та регістрові операнди. Два старші біти цього байта (поле *Mod*) і три його молодші біти (поле *R/M*) об’єднуються, формуючи поле з 5 біт із 32 можливими значеннями. З них 8 значень ідентифікують реєстрові операнди, а інші 24 значення визначають режими адресації. Решта 3 біти (поле *reg/opcode*) вказують регістр або містять три додаткові біти опкоду залежно від інструкції.
- **Байти зміщення адреси.** 0, 1, 2 або 4 байти забезпечують зсув адреси, що використовується при обчисленні адреси операнда.
- **Байти безпосереднього значення.** Якщо інструкція містить у собі не-опосередковане значення, воно міститься в останніх 1, 2 або 4 байтах інструкції.

Змінна довжина інструкцій *x*86 робить процес їхнього декодування досить складним. Крім того, в інструментів налагодження можуть виникнути труднощі з розбором послідовності інструкцій у зворотному порядку, що може знадобитися для відображення коду, що передує контрольній точці.

Ці труднощі виникають через те, що завершальна послідовність байтів у довгих інструкціях може сформувати повну допустиму інструкцію. Такі складнощі є помітною відмінністю від більш звичних форматів інструкцій, використовуваних в архітектурах RISC.

Мова асемблера x86

Мовою асемблера можна розробляти програми будь-якого рівня складності.

Однак більшість сучасних додатків значною мірою або повністю розробляють мовами високого рівня. Мову асемблера, як правило, використовують у тих випадках, коли виникає потреба в спеціалізованих інструкціях або потрібна екстремальна оптимізація, недосяжна за допомогою оптимізуючого компілятора.

Незалежно від мови, використовуваної під час розроблення додатків, увесь код зрештою має виконуватися у вигляді інструкцій процесора. Для повного розуміння того, як виконується код у комп'ютерній системі, немає іншого шляху, крім вивчення стану системи після виконання кожної окремої інструкції.

Гарним способом навчитися розуміти і працювати в цьому середовищі є написання деякого коду на асемблері.

Приклад мовою асемблера *x*86 у наступному лістингу являє собою закінчений додаток *x*86, який запускається з командної консолі Windows, друкує рядок тексту і потім завершує роботу.

```
.386
.model FLAT,C
.stack 400h


.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near extern exit:near

public main
main proc
    ; Друк повідомлення
    push offset message
    call printf

    ; Вихід з програми з кодом стану 0
    push 0
    call exit
main endp

.data
message db "Привіт, архітектор комп'ютерів!",0

end
```

Нижче наведено опис вмісту цього файлу Мовою асемблера.

- Директива `.386` вказує, що інструкції в цьому файлі слід інтерпретувати як інструкції для процесорів 80386 і пізніших поколінь.
- Директива `.model` FLAT,C визначає 32-розрядну плоску модель пам'яті та використання угод про виклик функцій мовою C.
- Директива `.stack` 400h визначає розмір стека в 400h (1024) байт.
- Директива `.code` вказує на початок виконуваного коду.
- Директиви `includelib` і `extern` посилаються на надані системою бібліотеки і функції в них, які будуть використовуватися програмою.
- Директива `public` вказує, що ім’я функції `main` є видимим ззовні.
- Рядки між `main proc` і `main endp` містять інструкції на мові асемблера, що складають функцію `main`.
- Директива `.data` вказує на початок пам’яті даних. Вираз `message db` визначає рядок повідомлення як послідовність байтів, за якою слідує нульовий байт.
- Директива `end` позначає закінчення програми.

Цей файл з ім’ям `hello_x86.asu` транслюється і компонується для одержання виконуваного файлу програми `hello_x86.exe` за допомогою наступної команди, яка запускає Microsoft Macro Assembler:

```
ml /Fl /Zi /Zd hello_x86.asm
```

Компоненти цієї команди:

- `ml` запускає асемблер (`ml.exe`);
- `/Fl` створює файл лістингу;
- `/Zi` включає в виконуваний файл налагоджувальну інформацію про імена, що використовуються в програмі;
- `/Zd` включає в виконуваний файл налагоджувальну інформацію про номери рядків;
- `hello_x86.asm` – це ім’я вихідного файлу мовою асемблера.

Нижче показано частину файлу лістингу hello_x86.lst, згенерованого асемблером.

```
                        .386
                        .model FLAT,C
                        .stack 400h
00000000                .code
```

```
                        includelib libcmt.lib
                        includelib legacy_stdio_definitions.lib

                        extern printf:near
                        extern exit:near

                        public main
00000000                main proc
                             ; Друк повідомлення
00000000 68 00000000 R       push offset message
00000005 E8 00000000 E       call printf
                             ; Вихід з програми з кодом стану 0
0000000A 6A 00 push 0
0000000C E8 00000000 E     call exit
00000011                  main endp

00000000                 .data
00000000 48 65 6C 6C 6F message db " Привіт, архітектор комп'ютерів!",0
         2C 20 43 6F 6D
         70 75 74 65 72
         20 41 72 63 68
         69 74 65 63 74
         21 00
```

У лівому стовпці цього лістингу відображаються зміщення адрес від початку функції main. У рядках, що містять інструкції, за зміщенням адреси слідує код операції. Адресні посилання в коді (наприклад, `offset message`) відображаються в лістингу у вигляді `00000000`, оскільки їх значення визначаються під час компонування, а не в ході трансляції, коли створюється цей лістинг.

Нижче наведено вихідні дані, що відображаються при запуску програми.

```
C:\>hello_x86.exe
Привіт, архітектор комп'ютерів!
```

Далі ми розглянемо розширення 32-розрядної архітектури x86 до 64-розрядної архітектури x64.

Архітектура та набір інструкцій x64

Початкова специфікація для процесорної архітектури, що розширює процесор *x*86 і набір інструкцій до 64 розрядів і отримала назву **AMD64**, була представлена компанією AMD в 2000 р. Перший процесор AMD64, Opteron, був випущений в 2003

р. Intel, виявивши своє відставання від AMD, розробила архітектуру, сумісну з AMD64, яка в результаті була названа **Intel 64**. Першим процесором Intel з 64-розрядною архітектурою став Xeon, представлений в 2004 р. Архітектурам AMD і Intel дали загальну назву ***x*86-64**, що відображає розвиток від x86 до 64 біт. У загальноприйнятому вживанні цей термін був скорочений до ***x*64**.

Перша версія Linux, що підтримує архітектуру x64, була випущена в 2001 р., задовго до появи перших процесорів x64. У Windows підтримка архітектури *x*64 була додана в 2005 р.

Процесори, що реалізують архітектури AMD64 і Intel 64, значною мірою сумісні на рівні набору інструкцій для програм користувацького режиму. Існує кілька відмінностей між цими архітектурами, найбільш значущою з яких є різниця в підтримці інструкцій Intel sysenter/sysexit та інструкцій AMD syscall/sysret, яку ми обговорювали раніше.

В цілому операційні системи та компілятори мов програмування справляються з цими відмінностями, тому вони зрідка стають предметом уваги розробників програмного забезпечення та систем. Розробники програмного забезпечення ядра, драйверів та асемблерного коду повинні враховувати ці відмінності.

Основні особливості архітектури x64 такі.

- x64 – це в основному сумісне 64-розрядне розширення 32-розрядної архітектури *x*86. Більшість програм, особливо додатків користувацького режиму, написаних для 32-розрядного середовища, повинні виконуватися без змін на процесорі, що працює в 64-розрядному режимі. 64-розрядний режим також називають **довгим режимом.**
- Вісім 32-розрядних регістрів загального призначення *x*86 в *x*64 розширені до 64 розрядів. На 64-розрядні регістри вказує префікс імені регістра R. Наприклад, в *x*64 розширений регістр *x*86 `EAX` називається `RAX`. Під компоненти регістрів *x*86 `EAX`, `AX`, `AH` і `AL` залишилися доступними і в *x*64.
- Покажчик інструкцій `RIP` розширений до 64 біт. Регістр прапорів `RFLAGS` також охоплює 64 біти, хоча його старші 32 біти зарезервовані. Молодші 32 біти регістра `RFLAGS` аналогічні `EFLAGS` в архітектурі *x*86.
- Було додано вісім 64-розрядних регістрів загального призначення, які отримали назви від `R8` , `R15`.
- 64-розрядні цілі числа підтримуються як власний тип даних.
- Процесори x64 зберігають можливість роботи в режимі сумісності з *x*86. Цей режим підтримує 32-розрядні операційні системи і дозволяє будь-якій програмі, створеній для *x*86, працювати на процесорах *x*64. У режимі сумісності з 32-розрядним середовищем 64- розрядні розширення недоступні.

Віртуальні адреси в архітектурі x64 мають розрядність 64 біти, підтримуючи адресний простір розміром 16 **ексабайт** (**Ебайт**), що еквівалентно 2^{64} байтам. Однак сучасні процесори AMD і Intel підтримують тільки 48 біт віртуального адресного простору. Це обмеження знімає апаратну складність процесора, зберігаючи при цьому підтримку до 256 **терабайт** (**Тбайт**) віртуального адресного простору. Процесори поточного покоління також підтримують максимум 48 біт фізичного адресного простору. Це дозволяє професору обробляти 256 Тбайт фізичної оперативної пам'яті, хоча сучасні материнські плати не підтримують кількість пристроїв DRAM, яка може знадобитися такій системі.

Набір регістрів архітектури x64

В архітектурі *x*64 розширення розміру регістрів *x*86 до 64 біт і додавання регістрів R8–R15 дають карту регістрів, показану на рис. 10.3.

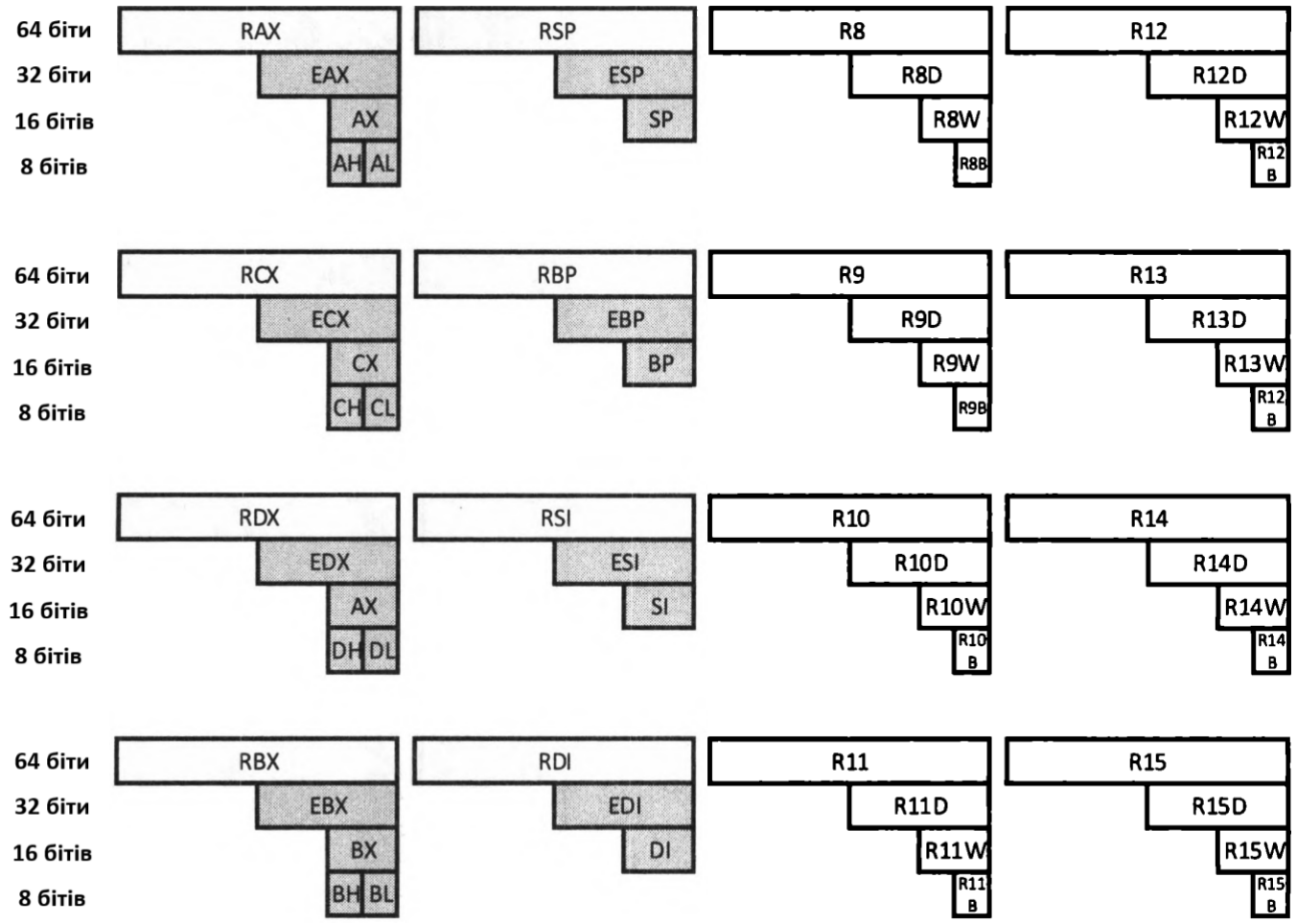


Рис 10.3. *Реєстри архітектури x64*

На рис. 10.3 регістри *x*86, описані в попередньому розділі (і присутні в x64), мають більш темний фон. При роботі в 64-розрядному режимі регістри x86 мають колишні імена і розміри.

Розширені 64-розрядні версії регістрів x86 мають імена, що починаються з літери R. Передбачена можливість доступу до менших частин нових регістрів 64-розрядної архітектури (`R8-R15`) за допомогою відповідної літери суфікса:

- суфікс **D** забезпечує доступ до молодших 32 бітів регістра: R11D;
- суфікс **W** забезпечує доступ до молодших 16 бітів регістра: R11W;
- суфікс **B** забезпечує доступ до молодших 8 бітів регістра: R11B.

На відміну від регістрів x86, нові регістри в архітектурі x64 мають дійсно загальне призначення і не виконують жодних спеціальних функцій на рівні інструкцій процесора.

Категорії та формати інструкцій x64

Архітектура *x*64 реалізує по суті той самий набір інструкцій, що і *x*86, але з розширеннями для 64-розрядного середовища. При роботі в 64-розрядному режимі архітектура *x*64 за замовчуванням використовує адреси розміром 64 біти і операнди розміром 32 біти. Новий байт префікса `rex` коду операції визначає використання 64-розрядних операндів.

Формат інструкцій *x*64 в пам’яті відповідає формату архітектури *x*86 з деякими незначними для наших цілей винятками. Додавання підтримки байта префікса `rex` є найбільш значною відмінністю від формату інструкцій *x*86. Зсуви адрес і безпосередні значення в деяких інструкціях можуть мати розрядність 64 біти на додаток до всіх інших варіантів розрядності, що підтримуються в *x*86.

Незважаючи на можливість визначити інструкції довжиною більше 15 байт, декодер інструкцій процесора видасть помилку загального захисту, якщо буде зроблена спроба декодувати інструкцію довжиною більше 15 байт.

Мова асемблера x64

Вихідний файл на асемблері *x*64 для програми hello схожий на версію цього ходу для *x*86, з деякими помітними відмінностями.

- Відсутня директива, що визначає модель пам’яті, оскільки в *x*64 існує тільки одна модель пам'яті.
- В **інтерфейсі програмування додатків** (application programming interface, API) Windows *x*64 використовується угода про виклик, яка зберігає перші чотири аргументи викликаної функції в регістрах `RCX`, `RDX`, `R8` і `R9` у зазначеному порядку. Цей підхід відрізняється від прийнятої за замовчуванням угоди про виклики *x*86, згідно з якою параметри розміщуються в стеках. Обидві бібліотечні функції, що викликаються цією програмою (`printf` і `exit`), отримують один аргумент, що передається через регістр `RCX`.

- Угода про виклик вимагає, щоб програма, що викликає функцію, виділяла простір стека для зберігання як мінімум тієї кількості аргументів, які передаються викликаній функції, з мінімальним резервуванням простору під чотири аргументи, навіть якщо число переданих аргументів менше. Оскільки стек росте в пам'яті вниз, для цього потрібно віднімання з покажчика стека. Таке резервування стека виконує інструкція `sub rsp, 40`. Після повернення з викликаної функції для скасування цього резервування зазвичай потрібна корекція покажчика стека. Наша програма для завершення виконання викликає функцію `exit`, що робить цей крок непотрібним.

Код 64-розрядної версії програми hello виглядає наступним чином:

```
.code
includelib libcmt.lib
includelib legacy_stdio_definitions.lib

extern printf:near
extern exit:near

public main
main proc
    ; Резервування місця в стеку
    sub    rsp, 40

    ; Друк повідомлення
    lea    rex, message
    call   printf

    ; Вихід з програми з кодом стану 0
    xor    rcx, rcx
    call   exit
main endp

.data
message db "Привіт, архітектор комп'ютерів!",0

end
```

Цей файл з іменем `hello_x64.asm` і транслюється, і компонується для отримання виконуваного файлу програми `hello_x64.exe` за допомогою наступного виклику Microsoft Macro Assembler (версія *x*64):

```
ml64 /Fl /Zi /Zd hello_x64.asm
```

Компоненти цієї команди:

- `ml64` запускає 64-розрядний асемблер;
- `/Fl` створює файл лістингу;
- `/Zi` включає в виконуваний файл налагоджувальну інформацію про використовувані в програмі іменах;
- `/Zd` включає в виконуваний файл налагоджувальну інформацію про номери рядків;
- `hello_x64.asu` – це ім'я вихідного файлу на мові асемблера.

Нижче показана частина файлу лістингу `hello_x64.lst`, згенерованого цією командою асемблера:

```
00000000                .code
                        includelib libcmt.lib
                        includelib legacy_stdio_definitions.lib

                        extern printf:near
                        extern exit:near

                        public main
00000000                main proc
                             ; Резервування місця в стеку
00000000  48/ 83 EC 28       sub    rsp, 40

                             ; Друк повідомлення
00000004  48/ 8D 0D          lea    rex, message
          00000000 R
0000000B  E8 00000000 E      call   printf

                            ; Вихід з програми з кодом стану 0
00000010  48/ 33 C9         xor    rcx, rcx
00000013 E8 00000000 E      call   exit
00000018                main endp


00000000                .data
00000000 48 65 6C 6C 6F message db " Привіт, архітектор комп'ютерів!",0
```

```
2C 20 43 6F 6D
70 75 74 65 72
20 41 72 63 68
69 74 65 63 74
21 00
```

Результат запуску цієї програми виглядає наступним чином:

```
C:\>hello_x86.exe
Привіт, архітектор комп'ютерів!
```

На цьому наше коротке знайомство з архітектурами *x*86 і *x*64 завершується. Але тут ще є чому повчитися – видання *«Intel 64 and IA-32 Architectures Software Developer's Manual», volumes 1–4 («Керівництво розробника програмного забезпечення для архітектур Intel 64 і IA-32", томи 1–4)* містить майже 5000 сторінок докладної документації по цим архітектурам.

У цьому розділі ми торкнулися лише невеликої частини цієї теми. Далі ми проведемо аналогічну оглядову екскурсію по 32-розрядних і 64-розрядних архітектурах ARM.

Архітектура набор інструментів 32-розрядних процесорів ARM

Архітектури ARM визначають сімейство RISC-процесорів, придатних для вирішення найрізноманітніших практичних завдань. Процесори, засновані на архітектурі ARM, є кращими в рішеннях, де потрібне поєднання високої продуктивності, низького енергоспоживання і компактних фізичних розмірів.

Архітектури ARM розробила британська компанія з виробництва напівпровідників і програмного забезпечення ARM Holdings, яка надає ліцензії на свою продукцію іншим компаніям, що виготовляють процесори. Архітектура ARM знаходить широке застосування у вигляді **систем на кристалі** (System-On-Chip, SoC), що поєднують процесор зі спеціалізованими апаратними засобами для реалізації таких функцій, як стільниковий радіозв'язок у смартфонах.

Процесори ARM використовуються в найрізноманітніших сферах – від кишенькових пристроїв на батарейках до суперкомп'ютерів. Ці процесори працюють як вбудовані процесори в критично важливих для безпеки системах, таких як антиблокувальні гальмівні системи автомобілів, і як процесори загального призначення в розумних годинниках, мобільних телефонах, планшетах, ноутбуках, настільних комп'ютерах і серверах. Станом на 2021 рік було вироблено понад 180 млрд процесорів ARM.

Процесори ARM – це повноцінні RISC-системи з великим набором регістрів загального призначення і однотактним виконанням більшості інструкцій. Стандартні інструкції ARM мають фіксований розмір 32 біти, хоча для систем, де пріоритет

віддається обсягу пам'яті, доступний окремий набір інструкцій змінної довжини **T32** (раніше його називали **Thumb**). Набір інструкцій T32 поєднує в собі 16- і 32-розрядні інструкції.

Процесори ARM поточного покоління підтримують як набір інструкцій ARM, так і T32 і можуть перемикатися між цими двома наборами в процесі роботи. Більшість операційних систем і додатків вважають за краще використовувати набір інструкцій T32, а не набір ARM, оскільки це допомагає підвищити щільність коду.

ARM – це **архітектура завантаження/збереження**, що вимагає завантаження даних з пам'яті в регістр, перш ніж з ними можуть бути виконані будь-які операції, наприклад обробка в АЛУ. Наступна інструкція зберігає результат назад в пам'ять. Це може здатися кроком назад у порівнянні з архітектурами *x*86 і x64, які працюють безпосередньо з операндами в пам'яті в рамках однієї інструкції, проте на практиці підхід із завантаженням/збереженням дозволяє з високою швидкістю виконувати кілька послідовних операцій над операндом відразу після його завантаження в один з багатьох регістрів процесора.

Процесори ARM підтримують можливість перемикання порядку байтів. Передбачено параметр конфігурації для вибору прямого або зворотного порядку байтів для багатобайтових значень. Значення за замовчуванням – прямий порядок байтів, оскільки цю конфігурацію зазвичай використовують операційні системи.

Архітектура ARM має вбудовану підтримку наступних типів даних:

- **byte** (байт) – 8 біт;
- **halfword** (півслово) – 16 біт;
- **word** (слово) – 32 біти;
- **doubleword** (подвійне слово) – 64 біти.

ЯКА ДОВЖИНА СЛОВА?



Між типами даних в архітектурі ARM і архітектурах *x*86 і *x*64 існує різниця, яка може викликати плутанину: в *x*86 і *x*64 слово має довжину 16 біт, а подвійне слово – 32 біти. В ARM слово складається з 32 біт, а подвійне слово – з 64 біт.

Процесори ARM підтримують вісім режимів роботи з різними привілеями виконання. Нижче наведено список цих режимів і їх скорочення:

- **користувацький режим** (user, USR);
- **режим супервізора** (supervisor, SVC);
- **режим швидкого переривання** (fast intemipt request, FIQ);

- **режим звичайного переривання** (intemipt request, IRQ);
- **режим моніторингу** (monitor, MON);
- **режим зупинки** (abort, ABT);
- **невизначений режим** (UND);
- **системний режим** (system, SYS).

Для цілей операційних систем і користувацьких додатків найбільш важливими рівнями привілеїв є USR і SVC. Два режими запиту переривання, FIQ і IRQ, використовуються драйверами пристроїв для обробки переривань.

У більшості операційних систем, що працюють на процесорах ARM, включаючи Windows і Linux, ядро функціонує в режимі ARM SVC, еквівалентному кільцю 0 в *x*86/*x*64. Режим USR в ARM еквівалентний кільцю 3 в *x*86/*x*64. Додатки, що працюють під управлінням Linux на процесорах ARM, використовують програмні переривання для запиту сервісів ядра, що передбачає перехід з режиму USR в режим SVC.

Архітектура ARM надає системні можливості, що виходять за межі можливостей головного процесора, за допомогою концепції співпроцесорів. Кожен співпроцесор реалізує спеціалізовану категорію функціональних можливостей, забезпечуючи підтримку головного процесора. У системі може бути реалізовано до 16 співпроцесорів, чотирьом з яких призначені заздалегідь визначені функції.

Співпроцесор 15 реалізує блок управління пам'яттю (MMU) та інші системні функції. При наявності в системі співпроцесор 15 повинен підтримувати опкоди інструкцій, набір регістрів і варіанти поведінки, встановлені для MMU. Співпроцесори 10 і 11 об'єднуються для реалізації операцій з плаваючою комою в процесорах, оснащених цією функцією. Співпроцесор 14 забезпечує підтримку функцій налагодження.

За минулі роки було випущено кілька версій архітектури ARM. В даний час широко використовується версія архітектури **ARMv8-A**, що підтримує 32- і 64-розрядні операційні системи і додатки. 32-розрядні додатки можуть працювати під управлінням 64-розрядної операційної системи ARMv8-A.

Практично всі висококласні смартфони і мобільні пристрої, що випускалися з 2016 р., спроектовані на базі процесорів або систем на кристалі (SOC), заснованих на архітектурі ARMv8-A. У наступному оповіданні основна увага буде приділена 32-розрядному режиму ARMv8-A. Відмінності 64-розрядного режиму ARMv8-A ми розглянемо в одному з наступних розділів цього розділу.

Набір регістрів ARM

У режимі USR архітектура ARM має шістнадцять 32-розрядних регістрів загального призначення: від `R0` до `R15`. Перші 13 регістрів мають дійсно загальне призначення, а решта три виконують наступні функції.