Лекція

з дисципліни: «Комп’ютерна схемотехніка та архітектура комп’ютерів»

на тему: «Покращені мікроархітектури»

ПЛАН ЛЕКЦІЇ

1. Довгі конвеєри.
2. Передбачення умовних переходів.
3. Суперскалярний процесор
4. SIMD
5. Багатопотоковість
6. Симетричні мультипроцесори

ЛІТЕРАТУРА

1. Рябенький В. М. Цифрова схемотехніка / Рябенький В. М., Жуйков В. Я., Гулий В. Д. – Львів: “Новий Світ-2000”, 2020. — 736 с.
2. Девід М. Харріс і Сара Л. Харріс. Цифрова схемотехніка та архітектура комп'ютера / пер. з англ. 2-е вид. - Morgan Kaufman, 2013. - 1621 с.
3. Мельник А.О. Персональні суперкомп’ютери: архітектура, проектування, застосування: монографія / А.О. Мельник, В.А. Мельник. – Львів: Видавництво Львівської політехніки, 2013. – 516 с.

Мікропроцесори використовують велику різноманітність прийомів для того, щоб виконувати програми швидше. Згадаймо, що час, необхідний для виконання програми, пропорційний періоду тактового сигналу, а також середній кількості тактів на команду (clock cycles per instruction - CPI). Таким чином, щоб збільшити продуктивність, необхідно або прискорити тактовий сигнал, або знизити CPI.

1.Довгі конвеєри

Найпростіший спосіб збільшити тактову частоту процесора - поділити конвеєр на більшу кількість стадій. Кожна стадія в цьому випадку містить менше логіки і, отже, може працювати швидше. В наш час нерідко можна побачити конвеєри в 10-20 стадій.

Максимальна кількість стадій конвеєра обмежена конфліктами в конвеєрі, накладними витратами на тимчасові регістри (тобто їх часом перевстановлення і утримання), вартістю розробки і виробництва. Довші конвеєри призводять до більшого числа залежностей усередині процесора. Деякі з залежностей можуть бути вирішені за допомогою байпаса (bypassing або forwarding), тобто передачі результату з більш пізніх стадій конвеєра в більш ранні безпосередньо, минаючи регістровий файл. Інші залежності вимагають призупинки (stalls), які збільшують CPI. Регістри, які знаходяться між стадіями конвеєра, призводять до накладних витрат через час їх попереднього встановлення (setup time) і затримки поширення (clk-to-Q delay), а також з-за зсуву фази тактового сигналу (clock skew), внаслідок чого додавання кожної наступної стадії дає все менший приріст продуктивності. Нарешті, додавання більшої кількості стадій збільшує вартість розробки і виробництва процесора.

В кінці 1990 і початку 2000 років зусиллями маркетологів вважалося, що чим вище частота мікропроцесора (1 / Tc), тим він кращий. Це призвело до того, що розробники стали використовувати дуже довгі конвеєри (від 20 до 31 стадії в Pentium 4), щоб максимально збільшити частоту, навіть якщо переваги в продуктивності були сумнівними. Енергоспоживання мікропроцесора пропорційно частоті тактового сигналу, а також збільшується через зростаюче число регістрів між стадіями, тому зараз, коли енергоспоживання таке важливе, число стадій в мікропроцесорах зменшується.

2. Передбачення умовних переходів

Теоретично CPI ідеального конвеєрного процесора має дорівнювати одиниці. Однією з основних причин більш високого CPI є простій процесора через неправильно передбачені умовні переходи (branch misprediction penalty). Зі збільшенням довжини конвеєра необхідність переходу з'ясовується в усе більш пізніх стадіях конвеєра. Таким чином, простій через неправильно передбачені переходи стає все більшим і більшим, так як конвеєр повинен бути очищений (flushed) від усіх команд, обраних після неправильно передбаченого переходу. Щоб вирішити / усунути / зменшити цю проблему, більшість конвеєрних процесорів використовують блок прогнозування умовних переходів (branch

predictor), що дозволяє з високою ймовірністю вгадати, чи варто здійснювати перехід.

Деякі умовні переходи відбуваються, коли програма доходить до кінця циклу (тобто в операторах for або while) і переходить до його початку для нової ітерації. Цикли часто виконуються багато разів, тому такі умовні переходи назад як правило виконуються. Найпростіший метод передбачення умовних переходів - це перевірити напрямок переходу і вважати, що перехід назад завжди буде виконаний. Такий метод називається статичним прогнозуванням переходів (static branch prediction), тому що він не залежить від історії виконання команд програми.

Переходи вперед важко передбачити без детального розуміння конкретної програми. Через це більшість процесорів використовують *динамічне прогнозування переходів* (dynamic branch predictors), яке використовує історію виконання програми для того, щоб передбачити, чи потрібно виконати перехід. Динамічне прогнозування переходів містять таблицю з останніми кількома сотнями (або тисячами) команд умовного переходу, виконаними процесором. Ця таблиця, яку іноді називають буфером цільових адрес розгалужень (branch target buffer), містить адреси переходів і інформацію про те, чи був перехід виконаний.

Однобітне динамічне прогнозування переходів (one-bit dynamic branch predictor) запам'ятовує, чи був перехід виконаний в минулий раз, і прогнозує, що в наступний раз відбудеться те ж саме. Поки цикл повторюється, блок прогнозування пам'ятає, що в минулий раз команду beq не було виконано, і прогнозує, що вона не буде виконана і в наступний раз. Цей прогноз залишається правильним аж до останньої ітерації, на якій перехід все-таки виконується.

Двохбітне динамічне прогнозування переходів вирішує цю проблему, використовуючи чотири стани: перехід точно виконається, перехід швидше виконається, перехід скоріше не виконається і перехід точно не виконається (strongly taken, weakly taken, weakly not taken, strongly not taken), як показано на Рис. 1.

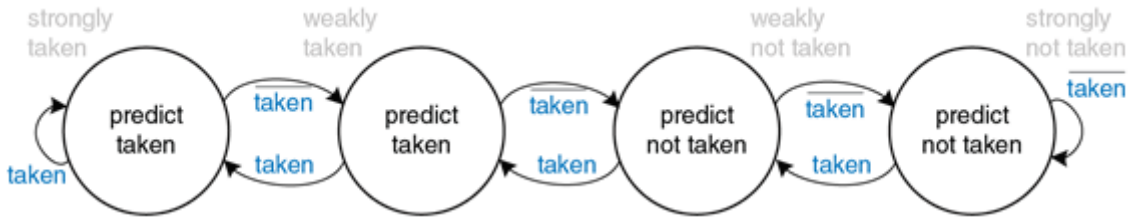

Рис.1 Діаграма станів

Поки цикл повторюється, блок прогнозування переходить в стан «точно не виконається» і залишається в ньому, прогнозуючи, що умовний перехід не буде виконаний і в наступний раз. Цей прогноз залишається вірним аж до останньої ітерації, на якій умовний перехід виконується і переводить блок прогнозування в стан «скоріше не виконається». Коли цикл почнеться знову, блок прогнозування переходів правильно передбачить, що перехід не повинен бути виконаний, і знову перейде в стан «точно не виконається».

Блок прогнозування переходів працює на стадії вибірки команд з пам'яті (стадія Fetch) конвеєра і використовується процесором, щоб визначити, яку команду вибирати на наступному такті. Коли блок прогнозування каже, що умовний перехід буде виконаний, процесор вибирає наступну команду з комірки пам'яті, адреса якої знаходиться в таблиці адрес переходів (branch target buffer). У цій таблиці зручно тримати адреси призначення як умовних, так і безумовних переходів, щоб уникнути очищення конвеєра від марно обраних команд, що настають за командою безумовного переходу.

3. Суперскалярний процесор

Скалярний процесор (scalar processor) в кожен момент часу здійснює обчислення тільки над однією порцією даних (число команд, що одночасно знаходяться в стадії Execute, а не в усьому конвеєрі). Векторний процесор (vector processor) працює над декількома порціями даних одночасно, але використовує для цього тільки одну команду. Суперскалярний процесор (superscalar processor) запускає на виконання (issues) кілька команд одночасно, кожна з яких оперує над однією порцією даних.

Тракт даних суперскалярного процесора містить кілька копій функціональних блоків, що дозволяє йому виконувати кілька команд одночасно. На Рис. 2 показана діаграма двоканального (2-way) суперскалярного процесора, який здійснює вибірку і виконання двох команд за такт. Тракт даних вибирає з пам'яті команд дві команди за раз. Він містить регістровий файл із шістьма портами, щоб читати чотири операнди і записувати назад два результати на кожному такті. Тракт даних також містить два АЛП і двохпортову пам'ять даних, щоб виконувати дві команди одночасно.

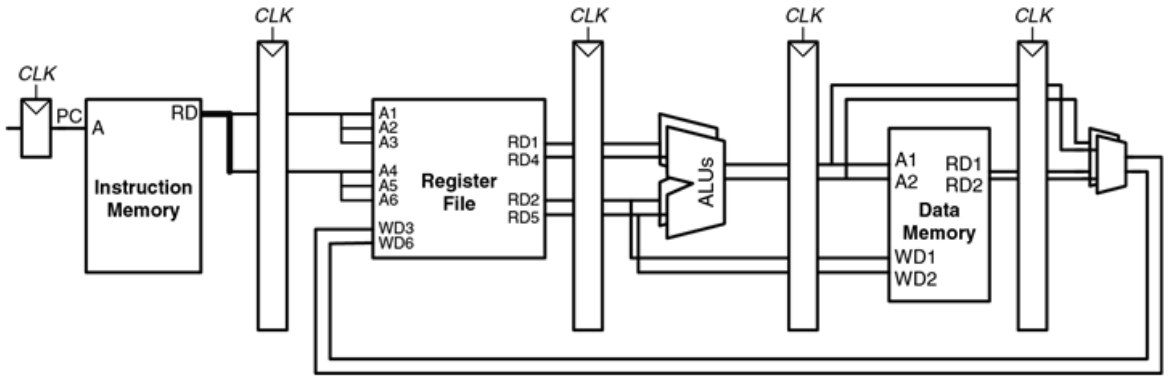

Рис. 2

На Рис. 3 наведена діаграма двоканального суперскалярного процесора, який виконує дві інструкції на кожному такті. В цьому випадку CPI процесора дорівнює 0,5. Розробники часто використовують величину, зворотну CPI - кількість команд на такт (instructions per cycle, IPC), яке для цього процесора становить 2,0.

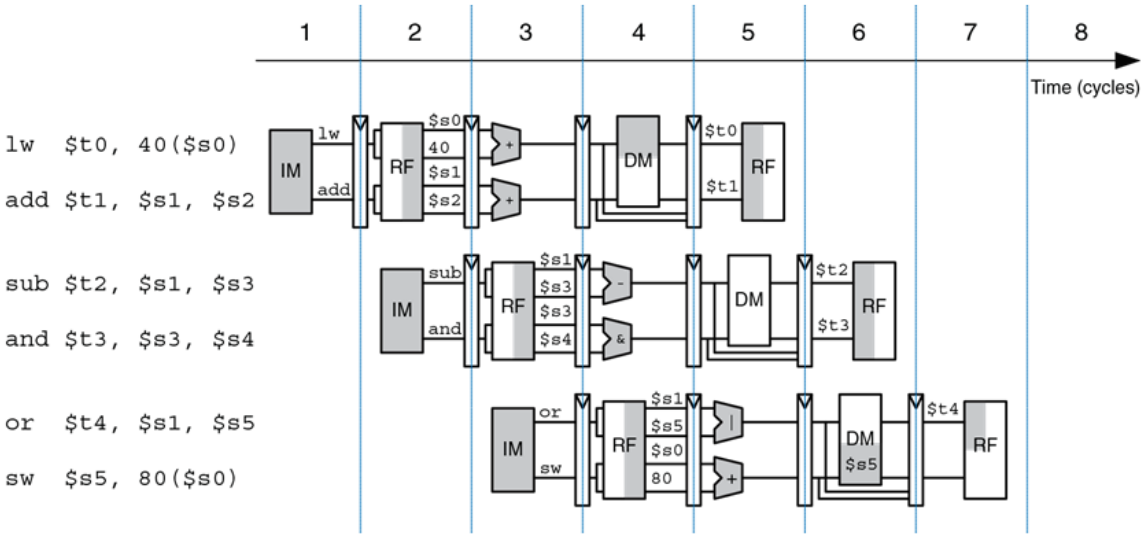

Рис.3

Комерційні суперскалярні процесори можуть бути три-, чотири- або навіть шестиканальним. Їм доводиться обробляти і конфлікти управління, що викликаються, наприклад, умовними переходами, і конфлікти даних. На жаль, в реальних програмах зустрічається багато залежностей, тому суперскалярні процесори з великою кількістю каналів рідко можуть використовувати всі свої функціональні блоки повністю. Більш того, велика кількість функціональних блоків і складності з організацією байпаса вимагають безлічі додаткових логічних елементів і споживають величезну кількість електроенергії.

4. SIMD

Одиночний потік команд, множинний потік даних (single instruction multiple data, SIMD) - це спосіб обробки даних, при якому одна команда обробляє безліч блоків даних паралельно. Типовим прикладом використання SIMD, особливо при обробці даних для комп'ютерної графіки, є виконання арифметичної операції над кількома невеликими незалежними блоками даних. Така обробка також називається *упакованою арифметикою* (packed arithmetic).

Наприклад, 32-бітний мікропроцесор може упаковувати чотири 8-бітних елементи (блоки) даних в одне 32-бітове слово. Команди додавання і віднімання упакованих чисел (packed add і packed sub відповідно) обробляють всі чотири елементи даних одночасно. На Рис. 4 показано, як команда додавання упакованих чисел підсумовує чотири пари 8-бітних чисел і отримує чотири результати. 32-бітове слово можна було б розділити і на два 16-бітних елементи. Для виконання команд упакованої арифметики необхідно модифікувати його АЛП так, щоб можна було відключати сигнали переносу між елементами даних. Тобто арифметичне переповнення при виконанні $a_0 + b_0$ не повинно впливати на результат виконання $a_1 + b_1$.

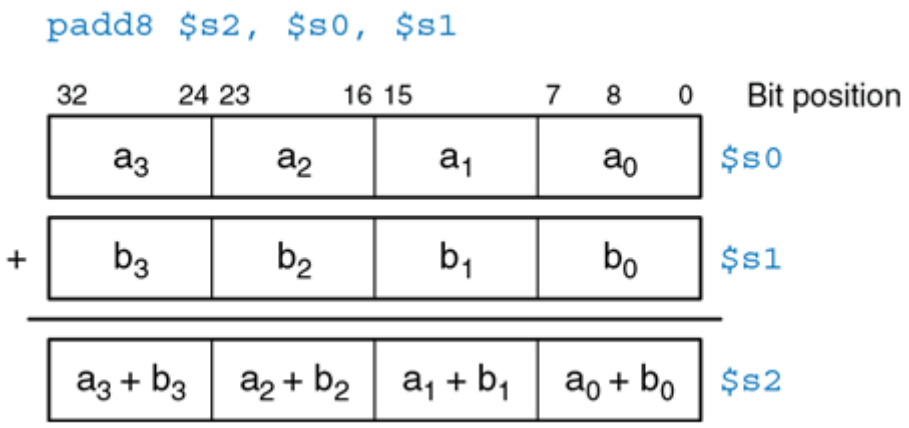

Рис. 4

Короткі елементи даних часто використовуються в додатках комп'ютерної графіки. Наприклад, піксель на цифровому фотознімку може використовувати по вісім біт для зберігання червоного, зеленого і синього компонентів кольору.

SIMD-команди для 64-бітних архітектур ще більш ефективні, так як в одне 64-бітове слово вони можуть упакувати вісім 8-бітних, чотири 16-бітних або два 32-бітних елементи.

5. Багатопотоковість

Багатопотоковість - це спосіб завантажити роботою процесор з великою кількістю функціональних блоків, навіть якщо у програми низький ILP (instruction level parallelism, паралелізм на рівні команд) або вона призупинена на час очікування даних з пам'яті.

Програма, яка виконується на комп'ютері, називається процесом (process). Комп'ютери можуть виконувати кілька процесів одночасно. Наприклад, на своєму ПК ви можете слухати музику і працювати в інтернеті, одночасно запустивши антивірус. Кожен процес складається з одного або більше потоків команд (threads), які теж виконуються одночасно. Наприклад, в текстовому редакторі один потік може обробляти набір тексту користувачем, другий потік в цей час може перевіряти орфографію, третій - друкувати документ на принтері. При такій організації користувачеві не потрібно чекати, поки закінчиться друк, щоб знову набирати текст. Степінь, до якої процес можна розділити на кілька одночасно виконуваних потоків, визначає рівень паралелізму на рівні потоків (thread level parallelism, TLP).

У звичайному процесорі одночасна робота потоків - це тільки ілюзія. Реально ж потоки виконуються процесором по черзі під управлінням операційної системи. Коли «зміна» одного потоку добігає кінця, ОС зберігає його архітектурний стан, завантажує з пам'яті архітектурний стан наступного потоку і передає йому управління. Ця процедура називається *перемиканням контексту* (context switching). До тих пір, поки процесор перемикається між потоками досить швидко, користувачеві здається, що всі потоки виконуються одночасно.

У багатопотокового процесора є кілька копій архітектурного стану, внаслідок чого кілька потоків можуть бути активні одночасно (те, що кілька процесів можуть бути активні одночасно, не означає, що всі вони можуть

виконуватися одночасно. Зазвичай в кожен момент часу виконується тільки один з них. Існують мікроархітектури, що об'єднують багатопотоковість і суперскалярність - ось вони можуть виконувати кілька команд з різних потоків одночасно на одному ядрі).

Багатопотоковість не покращує продуктивність окремого потоку, тому що вона не підвищує ILP. Тим не менш, вона покращує загальну пропускну здатність процесора, так як кілька потоків можуть більш повно використовувати ті ресурси процесора, які не використовуються при виконанні одного-єдиного потоку. Багатопотоковість відносно легко реалізувати, тому що потрібно додати тільки копії лічильника команд і регістрового файлу.

6.Симетричні мультипроцесори

Багатопроцесорна система (multiprocessor system), або просто мультипроцесор, складається з декількох процесорів і апаратури для з'єднання їх між собою. Найбільш часто зустрічається форма мультипроцесорності в комп'ютерних системах - це гомогенна мультипроцесорність (homogeneous multiprocessing), також звана симетричною мультипроцесорністю (symmetric multiprocessing, SMP), при якій два або більше однакових процесорів підключені до загальної основної пам'яті.

Процесори в мультипроцесорній системі можуть бути як кількома окремими мікросхемами, так і декількома ядрами (cores) всередині однієї мікросхеми. Приблизно в 2005 році розробники комп'ютерних мікроархітектури почалося розміщення по декількох копій процесора на одному кристалі; ці копії і називаються ядрами.

Мультипроцесори можна використовувати або для того, щоб виконувати більше потоків одночасно, або для того, щоб швидше виконувати один конкретний потік. Виконувати більше потоків одночасно досить просто - ці потоки можна просто розподілити між процесорами. На жаль, користувачам персональних комп'ютерів зазвичай не потрібно виконувати багато потоків - їм потрібно виконувати лише кілька потоків, але при цьому максимально швидко. Прискорення одного потоку за допомогою мультипроцесора - далеко не тривіальна задача. Щоб досягти цього, програміст повинен розділити один повільний потік на декілька менших потоків, які можна буде запустити на різних процесорах. Все ще більше ускладнюється, якщо процесорам потрібно обмінюватися даними. На сьогоднішній день ефективне використання великої кількості процесорних ядер - одна з головних проблем, що стоять перед розробниками комп'ютерів і програмістами.

7. Гетерогенні мультипроцесори

Гетерогенні мультипроцесори, які також називають асиметричними мультипроцесорами (asymmetric multiprocessors), використовують різні спеціалізовані мікропроцесори для різних завдань.

Гетерогенні мультипроцесори (heterogeneous multiprocessors покликані вирішити проблему хорошої середньої продуктивності з забезпеченням енергоефективності на широкому класі задач шляхом використання різних типів

процесорних ядер і / або спеціалізованої апаратури в одній системі. Кожна програма використовує ті ресурси, які дозволяють досягти або оптимальної роботи, або найкращого співвідношення продуктивності і енергоспоживання для конкретного додатку. Гетерогенні системи можуть приймати різні форми. Вони можуть включати в себе ядра з різними мікроархітектурами, що мають різне співвідношення енергоспоживання, продуктивності і займаної на кристалі площі. Наприклад, система може включати в себе як прості ядра з послідовним виконанням команд (in-order execution), так і більш складні суперскалярні ядра з позачерговим виконанням команд. Додатки, які можуть ефективно використовувати більш високопродуктивні, але менш енергоефективні ядра, отримують таку можливість, в той час як інші програми, які не можуть ефективно використовувати додаткову обчислювальну потужність суперскалярних ядер, будуть використовувати більш енергоефективні ядра з послідовним виконанням команд.

У такій системі ядра можуть використовувати як одну і ту ж систему команд, що дозволяє запускати додатки на будь-якому з ядер, так і різні системи команд, що відкриває додаткові можливості щодо пристосування ядер до поставленого завдання.

У гетерогенних систем є і недоліки. Вони складніші і в розробці, і в програмуванні, так як потрібно не тільки розробити різноманітні апаратні блоки, але і вирішити, коли і як найкращим чином використовувати різні ресурси системи. В кінцевому підсумку, і у гомогенних, і у гетерогенних систем є свої ніші. Гомогенні мультипроцесори підходять, наприклад, для великих центрів обробки даних, де не бракує задач з високим паралелізмом на рівні потоків. Гетерогенні системи більш продуктивні в разі більш різноманітною обчислювального навантаження і обмеженого паралелізму.