Лекція

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

на тему: «Конвеєрний процесор»

ПЛАН ЛЕКЦІЇ

1. Принцип конвеєризації.
2. Конвеєрний тракт даних.
3. Конвеєрний пристрій управління
4. Конфлікти

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.

1.Принцип конвеєризації

Ми розглянемо конвеєрний процесор, розділивши однотактний процесор на п'ять стадій. Таким чином, п'ять команд зможуть виконуватися одночасно, по одній в кожній стадії. Так як кожна стадія містить тільки одну п'яту від усієї логіки процесора, то частота тактового сигналу може бути майже в п'ять разів вище. Конвеєризація вимагає певних накладних витрат, так що в реальному житті пропускна здатність буде нижчою, ніж в ідеальному випадку, але в будь-якому випадку у конвеєризації так багато переваг, що всі сучасні мікропроцесори - конвеєрні.

Читання і запис в пам'ять і регістровий файл, а також використання АЛП, зазвичай привносять найбільші затримки в процесорі. Ми поділимо конвеєр на стадії таким чином, щоб кожна з них включала тільки одну з цих операцій. Стадії ми назвемо Fetch (вибірка), Decode (дешифрування), Execute (виконання), Memory (доступ до пам'яті) і Writeback (запис результатів). Вони схожі на п'ять етапів виконання команди lw в багатотактному процесорі. В стадії Fetch процесор читає команду з пам'яті команд. В стадії Decode процесор читає операнди з регітрового файлу і дешифрує команду, щоб встановити керуючі сигнали. В стадії Execute процесор виконує обчислення в АЛП. В стадії Memory процесор читає або пише в пам'ять даних. Нарешті, в стадії Writeback процесор, якщо потрібно, записує результат в регістровий файл.

На Рис. 1 наведена часова діаграма для порівняння однотактного і конвеєрного процесорів. По горизонтальній осі відкладено час, а по вертикальній - команди. У однотактному процесорі, показаному на Рис. 1 (a), перша команда вибирається з пам'яті в момент часу 0; далі процесор читає операнди з регістрового файлу; далі АЛП виконує необхідні обчислення.

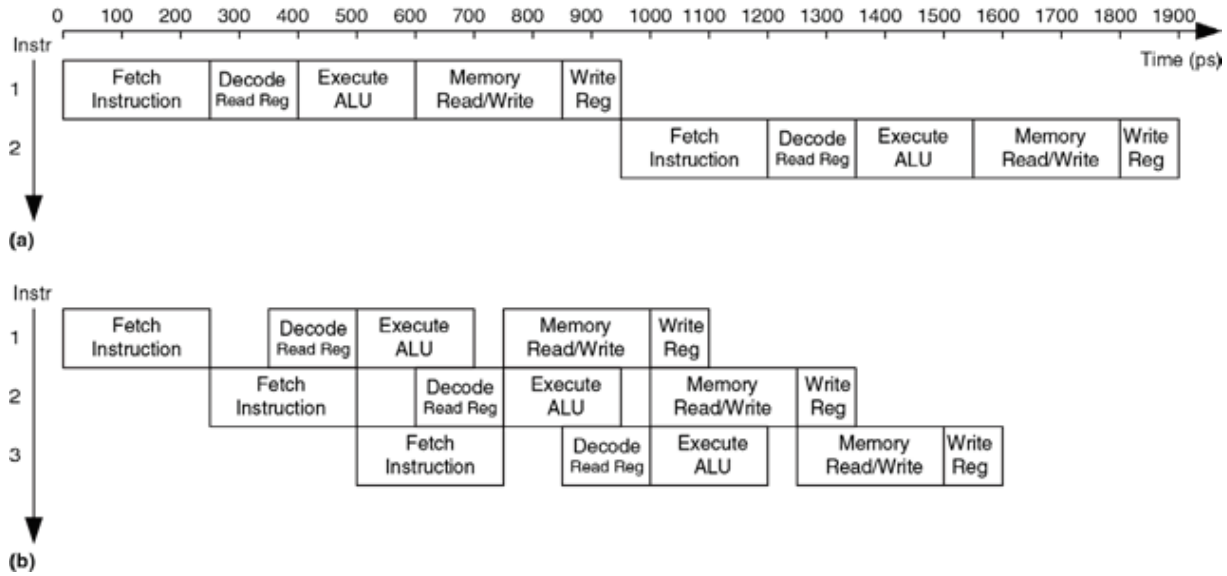

Рис.1 Часова діаграма однотактного (a) і конвеєрного (b) процесорів

Далі відбувається доступ до пам'яті даних, і результат записується в регістровий файл через 950пс. Виконання другої команди починається після того, як закінчена перша. Таким чином, як видно з діаграми, однотактний процесор забезпечує латентність команд, рівну 250 + 150 + 200 + 250 + 100 = 950

пс, і пропускну здатність, що дорівнює одній команді за 950 пс (1,05 мільярда команд в секунду).

У конвеєрному процесорі, наведеному на Рис. 1 (b), довжина стадії дорівнює 250 пс і визначається самою повільною стадією - стадією доступу до пам'яті (Fetch або Memory). У початковий момент часу перша команда вибирається з пам'яті. Через 250 пс, перша команда потрапляє в стадію Decode, а друга команда вибирається з пам'яті. Коли минуло 500 пс, перша команда починає виконуватися, друга потрапляє в стадію Decode, а третя вибирається. І так далі, поки всі команди не завершаться. Латентність команд становить 1100 пс. Пропускна здатність - одна команда за 250 пс (чотири мільярди команд в секунду). Латентність команд в конвеєрному процесорі трохи більше, ніж в однотактному, тому що стадії конвеєра не ідеально збалансовані, тобто не містять абсолютно однакову кількість логіки. Аналогічно, пропускна здатність процесора з п'ятистадійним конвеєром не в п'ять разів більша, ніж у однотактного. Проте, збільшення пропускної здатності досить значне.

На Рис. 2 показано абстрактне уявлення працюючого конвеєра, що ілюструє просування команд по конвеєру. Кожна стадія зображена картинкою, що містить головний компонент стадії - пам'ять команд (instruction memory, IM), читання регістрового файлу (register file, RF), АЛП, пам'ять даних (DM) і запис в регістровий файл (writeback). Якщо дивитися на рядки, то можна дізнатися, на якому такті команда знаходиться в тій чи іншій стадії.

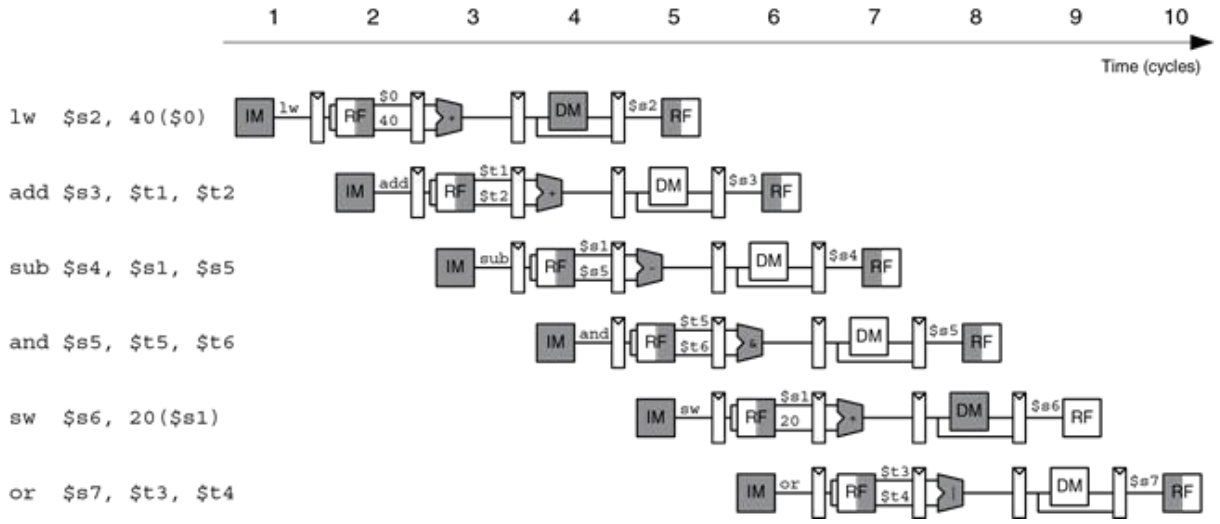

Рис.2

Стадії зафарбовані сірим, якщо вони використовуються. Наприклад, пам'ять даних використовується командами lw на четвертому такті і sw на восьмому. Пам'ять команд і АЛП використовуються на кожному такті. Результат записується в регістровий файл усіма командами, крім sw. У конвеєрному процесорі значення записуються в регістровий файл в першій частині такту, а читаються в другій, що так само зазначено на рисунку. У цьому випадку дані можуть бути записані і потім прочитані назад за один і той же такт.

Головна проблема в конвеєрних системах - вирішення конфліктів (hazards), які виникають, коли результати однієї з команд потрібні для виконання наступної команди до того, як перша завершиться. Наприклад, якби команда add

на Рис. 2 використовувала регістр $ s2 замість $ t2, то виник би конфлікт, тому що регістр $ s2 ще не був би записаний командою lw в той момент, коли команда add повинна була його прочитати. Існують такі методи вирішення конфліктів: пересилання даних через байпас (bypassing, або forwarding), призупинення (stalls) і скидання (flushes).

2. Конвеєрний тракт даних

Конвеєрний тракт даних можна отримати, розрізавши однотактний тракт даних на п'ять стадій, розділених регістрами (pipeline registers). На Рис. 3 показаний конвеєрний тракт даних, поділений на п'ять стадій шляхом вставки в нього чотирьох регістрів. Назви стадій і кордони між ними показані синім кольором. До всіх сигналів доданий суфікс (F, D, E, M або W), що показує, до якої стадії вони відносяться.

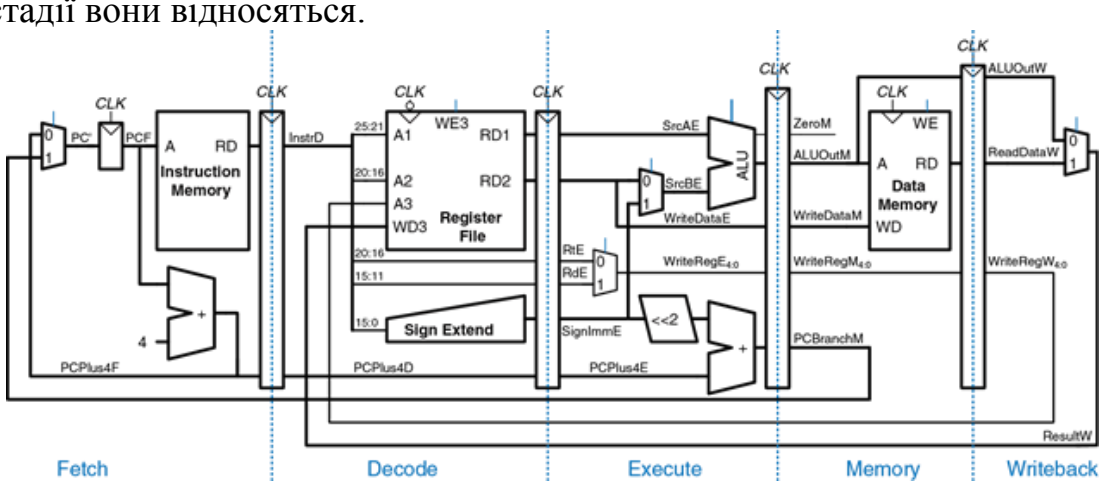

Рис.3

Регістровий файл - особливий в тому сенсі, що процесор читає з нього в стадії Decode, а пише в стадії Writeback. Тому, незважаючи на те, що на рисунку він знаходиться в стадії Decode, але адреса і дані для запису приходять зі стадії Writeback. Цей зворотний зв'язок буде приводити до конфліктів конвеєра.

Одна маленька, але надзвичайно важлива проблема організації конвеєрної обробки даних - це те, що всі сигнали, що відносяться до конкретної команди, повинні обов'язково просуватися по конвеєру одночасно один з одним, в унісон.

3. Конвеєрний пристрій управління

Пристрій управління формує керуючі сигнали на основі полів opcode і funct, присутніх в кожній команді (Instr31: 26 і Instr5: 0 відповідно). Конвеєрний процесор використовує ті ж сигнали, що і однотактний процесор, тому використовує такий же пристрій управління. В стадії Decode він, в залежності від полів opcode і funct команди, формує керуючі сигнали. Ці сигнали повинні бути конвеєризовані точно так, як і тракт даних, щоб залишатися синхронними з командою, яка переміщується з однієї стадії в іншу.

Повністю конвеєрний процесор з пристроєм управління показаний на Рис. 4.

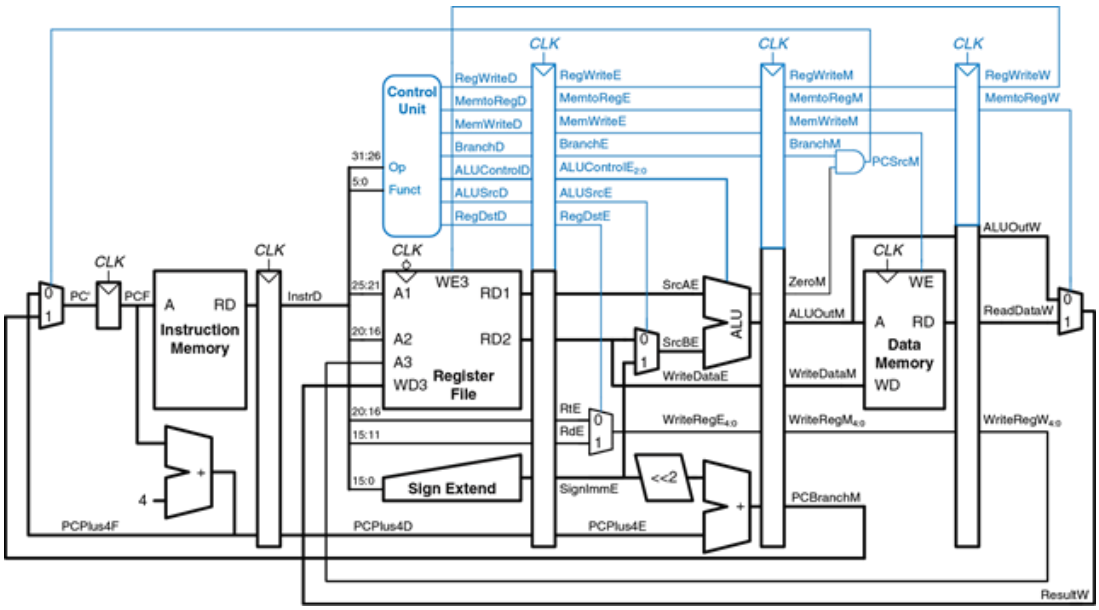

Рис.4

Коли сигнал ALUOp дорівнює 00 або 01, АЛП має додавати або віднімати відповідно. Коли він дорівнює 10, то значення ALUControl має визначатися полем funct. Для команд типу R перші два біта поля funct завжди рівні 10

4. Конфлікти

У конвеєрному процесорі виконуються кілька команд одночасно. Коли одна з них залежить від результатів іншої, ще не завершеної команди, то кажуть, що стався конфлікт (hazard) в конвеєрі.

Процесор може читати і писати в регістровий файл за один такт. Запис відбувається в першій частині такту, а читання - у другій, так що значення в регістр можна записати і потім прочитати назад за один такт, і це не призведе до конфлікту.

На Рис. 5 показаний конфлікт, який виникає, коли одна команда пише в регістр ($ s0), а наступна команда читає з нього. Це називається конфліктом читання після запису (read after write, RAW). Команда add записує результат в $ s0 в першій частині п'ятого такту, проте команда and читає $ s0 на третьому такті, тобто отримує невірне значення. Команда or читає $ s0 на четвертому такті, і теж отримує невірне значення. Команда sub читає $ s0 в другій половині п'ятого такту, тобто нарешті отримує коректне значення, яке було записано в регістр в першій половині п'ятого такту. Всі наступні команди також прочитають коректне значення з $ s0. Як видно з діаграми, конфлікт в конвеєрі виникає тоді, коли команда записує значення в регістр і хоча б одна з наступних двох команд читає його. Якщо не вжити заходів, конвеєр вирахує неправильний результат.

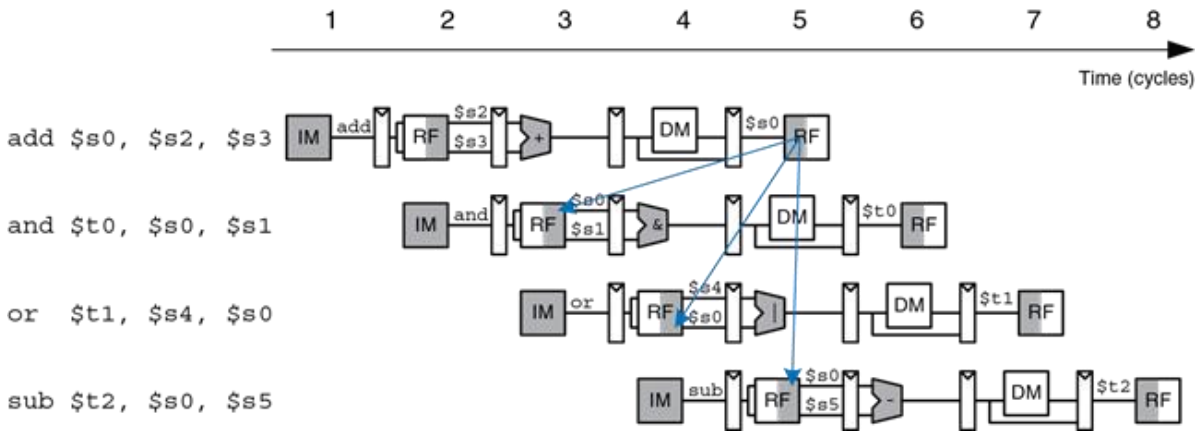

Рис. 5 Конфлікти в конвеєрі

Однак якщо поглянути, то результат команди add обчислюється в АЛП на третьому такті, а команді and він потрібен лише на четвертому. В принципі, ми могли б переслати результат виконання першої команди другій до того, як він буде записаний в регістровий файл, вирішивши конфлікт читання після запису без необхідності призупиняти конвеєр. Частина тракту даних, що забезпечує таку пересилку, називається байпасом (bypass).

Конфлікти можна розділити на конфлікти даних (data hazards) і конфлікти управління (control hazards). Конфлікт даних відбувається, коли команда намагається прочитати з регістра значення, яке ще не було записано попередньою командою. Конфлікт управління відбувається, коли процесор вибирає з пам'яті наступну команду до того, як стало ясно, яку саме команду треба вибрати. Для вирішення конфліктів в процесорі присутній блок вирішення конфліктів (hazard unit), який буде виявляти і вирішувати конфлікти таким чином, щоб процесор виконував програми коректно.

Вирішення конфліктів пересиланням через байпас

Деякі конфлікти даних можна вирішити шляхом пересилання результату через байпас (bypassing, також використовується термін forwarding) зі стадій Memory або Writeback в команду, яка очікує цей результат і знаходиться в стадії Execute. Щоб організувати байпас, знадобиться додати мультиплексори перед АЛП. Тепер операнд можна отримати або з регістрового файлу, або безпосередньо зі стадій Memory або Writeback, як показано на Рис. 6.

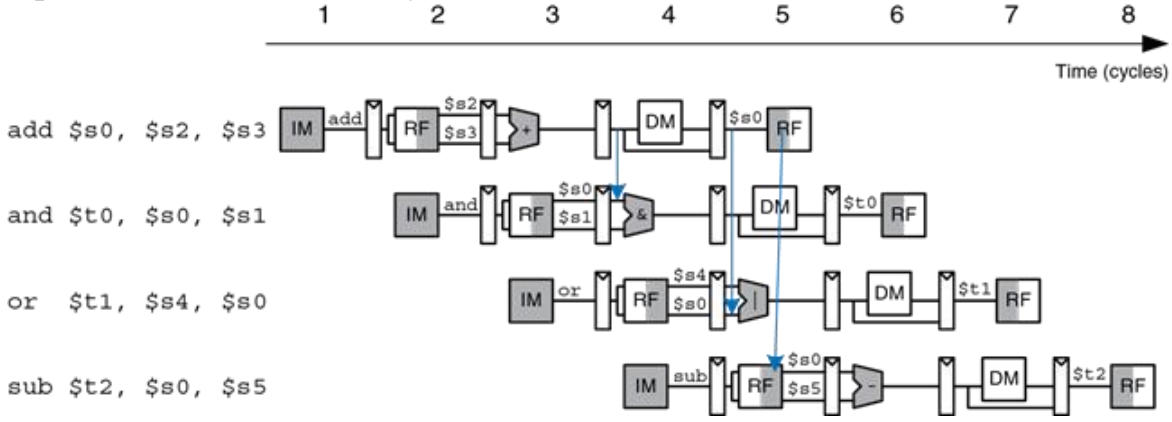

Рис. 6 Пересилка даних через байпас

Таким чином, на четвертому такті $ s0 пересилається через байпас з стадії Memory, де знаходиться команда add, в стадію Execute, де знаходиться команда

and, якій потрібен результат виконання add. На п'ятому такті $ S0 пересилається з стадії Writeback, де тепер знаходиться команда add, в стадію Execute, де знаходиться очікує її результату команда or.

Пересилання даних через байпас необхідна, якщо номер будь-якого з регістрів операндів команди, що знаходиться в стадії Execute, однаковий з номером регістра результату команди, що знаходиться в стадії Memory або Writeback. На Рис. 7 показаний модифікований конвеєрний процесор з байпасом.

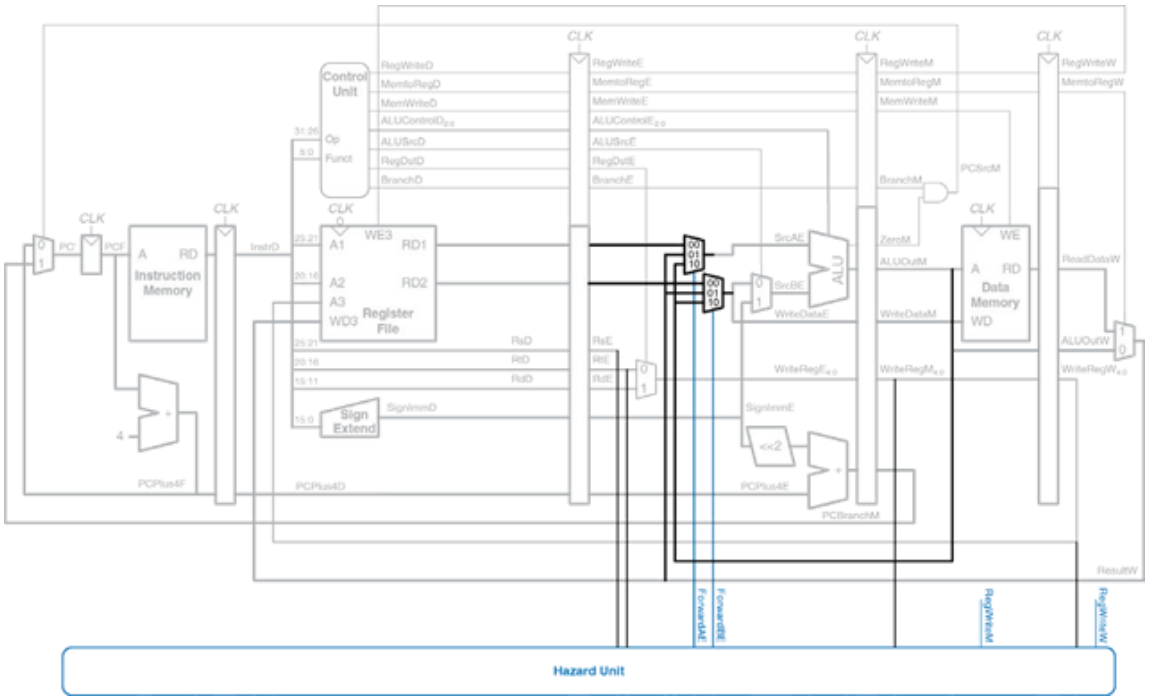

Рис. 7

У нього є блок виявлення конфліктів і два нових мультиплексори. Блок виявлення конфліктів отримує на свій вхід номери регістрів, що зберігають операнди команди, яка знаходиться в стадії Execute, а також номери регістрів результатів команд, які перебувають в стадіях Memory і Writeback. Також йому необхідні сигнали RegWrite зі стадій Memory і Writeback. Ці сигнали показують, чи потрібно насправді писати результат в регістр чи ні.

Блок виявлення конфліктів управляє мультиплексорами байпасу (forwarding multiplexers), які визначають, чи взяти операнди з регістрового файлу або переслати їх безпосередньо зі стадії Memory або Writeback. Якщо в одній з цих стадій відбувається запис результату в регістр і номер цього регістра збігається з номером регістру операнда наступної команди, то використовується байпас.

Вирішення конфліктів даних призупиненнями конвеєра

Пересилання даних через байпас може вирішити конфлікт при читанні після запису, тільки якщо результат обчислюється в стадії Execute, тому що тільки в цьому випадку його можна відразу переслати в стадію Execute наступної команди. На жаль, команда lw не може прочитати дані раніше, ніж в кінці стадії Memory, тому її результат не можна переслати в стадію Execute наступної команди.

Ця проблема показана на Рис. 8. Команда lw отримує дані з пам'яті в кінці четвертого такту. Однак команді and ці дані потрібні як операнд вже на самому початку четвертого такту. Пересилання даних через байпас не допоможе вирішити цей конфлікт.

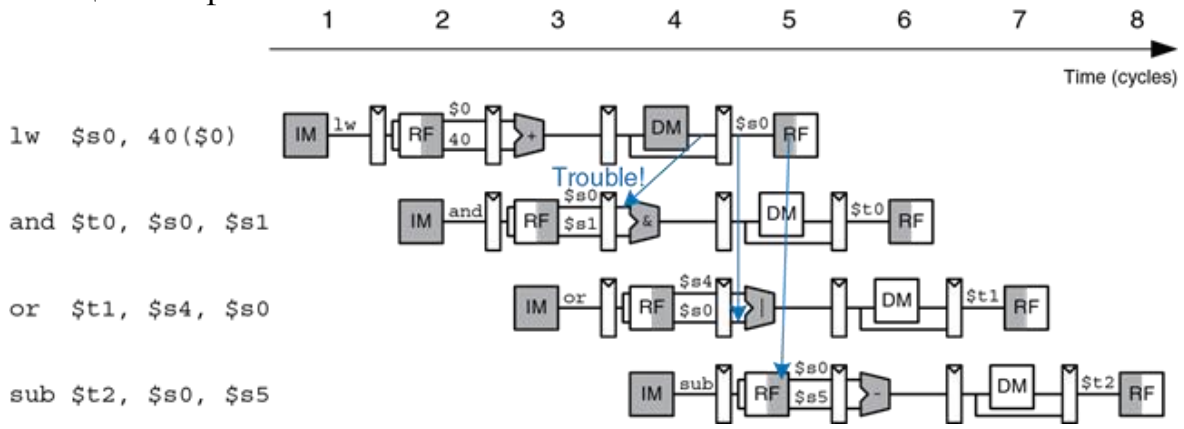

Рис.8

Альтернативне рішення - призупинити конвеєр, затримавши всі операції до тих пір, поки дані не стануть доступні. На Рис. 9 показана призупинення залежної команди (and) в стадії Decode. Команда and потрапляє в цю стадію на третьому такті і залишається там на четвертому такті. Наступна команда (or) повинна, відповідно, залишатися в стадії Fetch протягом третього і четвертого тактів, так як стадія Decode зайнята.

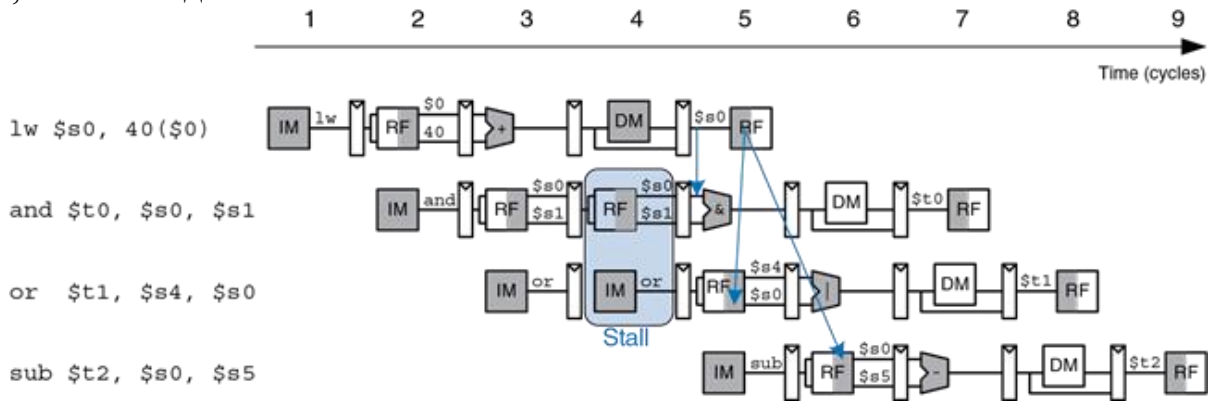

Рис. 9

На п'ятому такті результат команди lw можна через байпас переслати зі стадії Writeback в стадію Execute, де буде знаходитися команда and. На цьому ж такті операнд $ s0 команди or може бути прочитаний прямо з регістрового файлу, без будь-якого пересилання даних.

Стадію конвеєра можна призупинити, якщо заборонити оновлення регістра, що знаходиться між цією і попередньою стадіями. Як тільки будь-яка стадія призупинена, всі попередні стадії теж повинні бути призупинені, щоб жодна з команд не пропала. Регістр, який розташовується одразу після призупиненої стадії, повинен бути очищений, щоб «сміття» не потрапило в конвеєр. Призупинення конвеєра погіршують продуктивність, тому повинні використовуватися тільки при необхідності.

Вирішення конфліктів управління

Виконання команди beq призводить до конфлікту управління: конвеєрний процесор не знає, яку команду вибрати наступною, оскільки в цей момент ще не ясно, чи потрібно буде виконати умовний перехід чи ні.

Один із способів вирішити цей конфлікт - призупинити конвеєр до тих пір, поки не буде прийнято потрібне рішення (тобто до тих пір, поки не буде вирахувано сигнал PCSrc). Рішення приймається в стадії Memory, так що для кожної команди умовного переходу доведеться призупиняти конвеєр на три такти. Таке рішення спричинило негативні наслідки для продуктивності системи.

Є й альтернативний спосіб - передбачити, чи буде виконаний умовний перехід чи ні, і почати виконувати команди, грунтуючись на цьому. Як тільки умова переходу буде визначена, процесор може перервати ці команди, якщо передбачення було невірним.

На Рис. 10 показаний конвеєрний процесор з раннім обчисленням умов переходів, здатний вирішувати конфлікти управління. У стадію Decode доданий компаратор, туди ж перенесений і логічний елемент I, використовуваний для отримання сигналу PCSrc, так що тепер PCSrc формується в стадії Decode, а не Memory, як раніше. Щоб адреса переходу PCBranch була визначена вчасно, відповідний суматор теж повинен бути перенесений в стадію Decode. Для того щоб очистити стадію Decode від помилково обраної команди в разі неправильно передбаченого переходу, потрібно додати вхід синхронного скидання для тимчасового регістра, що знаходиться між стадіями Fetch і Decode, і підключити його до сигналу PCSrcD.

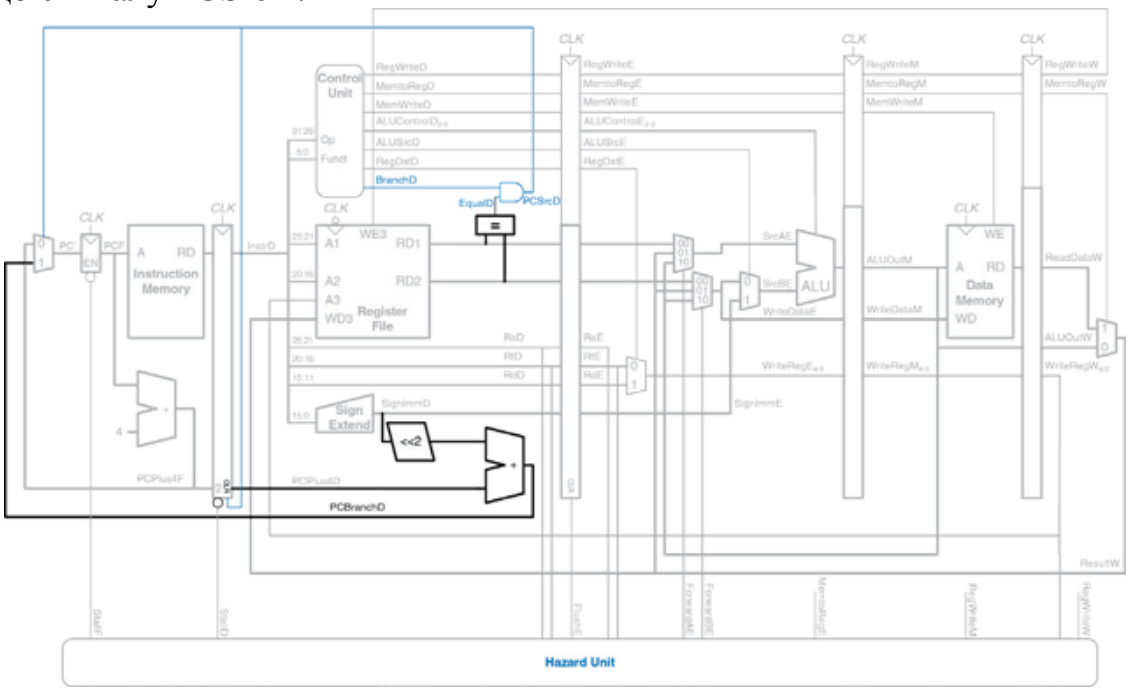

Рис. 10

Конфлікти управління вирішуються шляхом передбачення того, яка саме команда повинна бути обрана, і очищенням конвеєра від помилково вибраних команд в разі, якщо передбачення не збулося. Кількість команд, від яких доведеться очистити конвеєр в разі неправильно передбаченого переходу,

мінімізується шляхом переміщення логіки обчислення умови переходів на початок конвеєра.

На Рис. 11 показаний конвеєрний процесор, здатний вирішити всі наявні конфлікти.

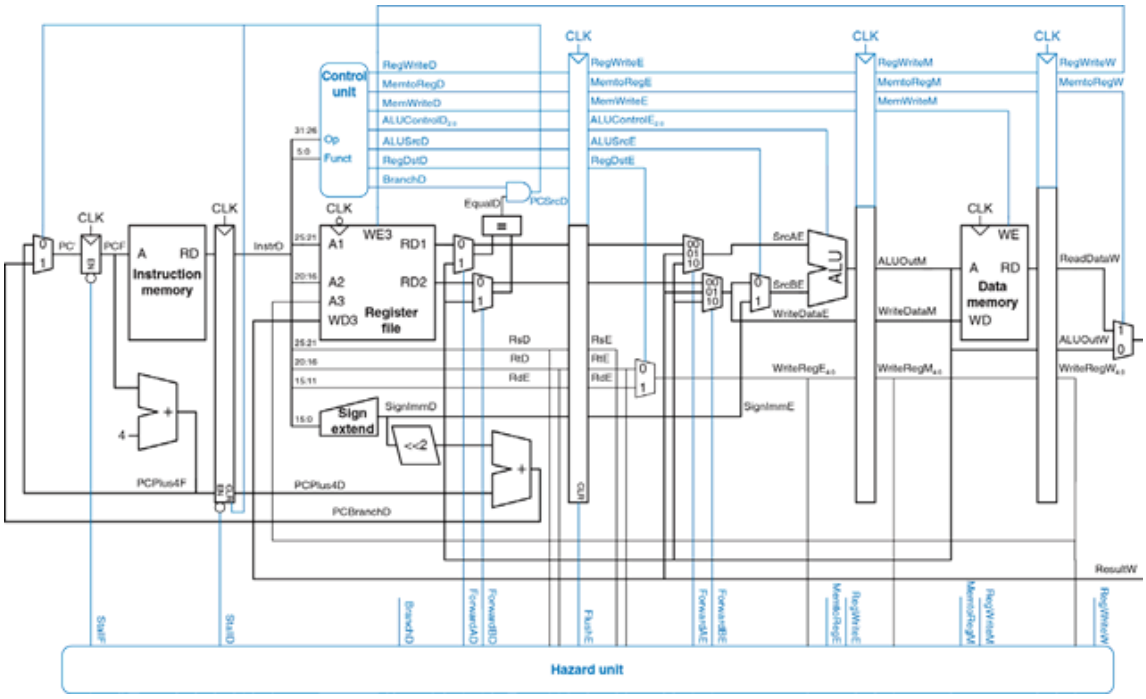

Рис. 11