

Розділ 5

Запобігання конфліктам в конвеєрі команд

У попередньому розділі були розглянуті структури процесора, які забезпечують суміщений в часі режим виконання кількох команд, коли вони є незалежними одна від одної. Це суміщення називається конвеєрним виконанням команд. У даному розділі буде розглянуто ряд методів підвищення ефективності конвеєрного виконання команд. Потреба в цьому викликана тим, що при реалізації конвеєрного виконання команд виникають ситуації, які перешкоджають виконанню чергової команди з потоку команд в призначенному для неї такті. Такі ситуації називаються конфліктами, або ризиками. Конфлікти знижують продуктивність конвеєра, яка могла б бути досягнута в ідеальному випадку. Більше того, конфлікти можуть звести наївець всі затрати на створення конвеєра команд.

Існує три класи конфліктів:

- Структурні конфлікти, які виникають з причини браку ресурсів, коли апаратні засоби не можуть підтримувати всі можливі комбінації команд в режимі одночасного виконання з перекриттям.
- Конфлікти за даними, що виникають у разі, коли виконання наступної команди залежить від результату виконання попередньої команди.
- Конфлікти керування, які виникають при конвеєризації команд передачі керування, які змінюють значення лічильника команд.

Конфлікти в конвеєрі призводять до необхідності призупинення виконання команд. Звичайно, якщо призупиняється виконання якої-небудь команди в конвеєрі, то виконання всіх наступних за нею команд також призупиняється і, зрозуміло, під час призупинення не вибирається жодна нова команда. При цьому команди, передуючі призупиненню, можуть продовжувати виконуватися.

Спочатку будуть розглянуті методи, що дозволяють знизити вплив структурних конфліктів, а також конфліктів за даними та конфліктів керування, а потім питання розширення можливостей комп'ютера по використанню паралелізму, закладеного в програмах. Далі буде проведено аналіз сучасних технологій компіляторів, які використовуються для збільшення ступеня паралелізму рівня команд.

5.1. Структурні конфлікти

Конвеєризація виконання команд в комп'ютері вимагає значних додаткових затрат обладнання, особливо якщо забезпечується безконфліктне виконання в конвеєрі всіх можливих комбінацій команд. Якщо яка-небудь комбінація команд не може бути вико-

нана в конвеєрному режимі через причини браку ресурсів, то говорять, що в комп'ютері є структурний конфлікт.

Існує дві групи структурних конфліктів. До першої групи належать структурні конфлікти, які виникають через потребу порушення тактової частоти роботи конвеєра. До другої групи належать структурні конфлікти, які виникають у зв'язку з необхідністю очікування на звільнення ресурсів комп'ютера.

Найбільш типовим прикладом комп'ютерів, у яких можлива поява структурних конфліктів першої групи, є комп'ютери з не повністю конвеєрними функціональними пристроями. Час виконання операції в такому пристрої може складати декілька тактів синхронізації конвеєра. В цьому випадку послідовні команди, які використовують даний функціональний пристрій, не можуть надходити до нього в кожному такті. Прикладом такого не повністю конвеєрного функціонального пристрою може бути пристрій додавання-множення на основі одного суматора, в якому додавання виконується за один такт, а множення з метою економії обладнання реалізується шляхом ітераційного виконання операції додавання. Такий підхід є виправданим при нечастому виконанні операції множення.

Можлива також інша, але досить подібна за впливом на конвеєр ситуація, коли в останньому включений пристрій, затримка в якому більша одного такту конвеєра. Наприклад, це може бути паралельний однотактовий поділювач двійкових чисел, затримка в якому перевищує такт конвеєра, а його конвеєризація є невиправдано дорогою із-за причини маловживаності цієї операції.

Поява структурних конфліктів першої групи викликає необхідність призупинення роботи конвеєра до закінчення роботи функціонального пристрою, звернення до якого викликало конфлікт. Кількість тактів простою конвеєра рівна відношенню часу виконання операції в функціональному пристрої до величини такту конвеєра. На рис. 5.1 наведено діаграму виконання на симулаторі WinDLX нижче приведеного фрагмента деякої програми

```
addf f6,f5,f4
xor r1,r3,r4
and r5,r6,r7,
```

у якій виник структурний конфлікт першої групи, викликаний наявністю в програмі команди додавання чисел з рухомою комою addf. Фаза виконання EX цієї команди є довгою в три рази, ніж такт роботи конвеєра. Тому фази MEM команд addf та and співпадають, тобто вимагається одночасний доступ до ресурсів фази MEM, що і стало причиною конфлікту. Через те виконання команди and призупиняється (Stall – зупинка). Час призупинення визначається алгоритмом та структурою пристрою додавання чисел з рухомою комою.

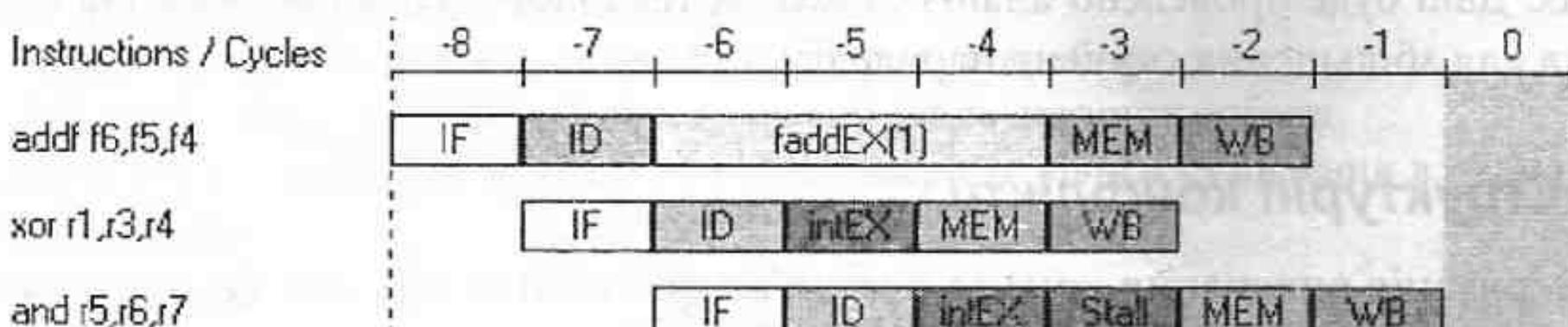


Рис. 5.1. Структурний конфлікт першої групи

Друга група структурних конфліктів пов'язана з недостатньою кількістю деяких ресурсів (функціональних блоків, портів і т. д.), що перешкоджає виконанню довільної послідовності команд в конвеєрі без його призупинення. Наприклад, процесор може мати тільки один порт запису в реєстрову пам'ять, але при певних обставинах конвеєру може виникнути необхідність виконати два записи в реєстрову пам'ять в одному такті. Це призводить до структурного конфлікту. Коли послідовність команд натрапляє на такий конфлікт, виконання однієї з команд призупиняється до тих пір, поки не стане доступним необхідний пристрій.

На рис. 5.2 показано приклад такого конфлікту для наступного фрагмента програми:

cvti2f f1, f1

cvti2f f2, f2

multf f5, f4, f8

multf f6, f3, f4.

Структурний конфлікт виникає на такті -8, коли відбувається звернення до пристрою множення для виконання фази EX команди multf f6,f3,f4. В цей час пристрій множення зайнятий виконанням фази EX попередньої команди множення, яка триває 5 тактів. Щоб зняти цей конфлікт, потрібно просто призупинити конвеєр на 4 такти, поки відбувається одночасне звернення до пристрою множення (S-stall – структурна зупинка). Подібне призупинення часто називається конвеєрною булькою, оскільки булька проходить по конвеєру, займаючи місце, але не виконуючи ніякої корисної роботи. Потрібно відзначити, що якби множення виконувалось за один такт, структурний конфлікт не виникнув би.

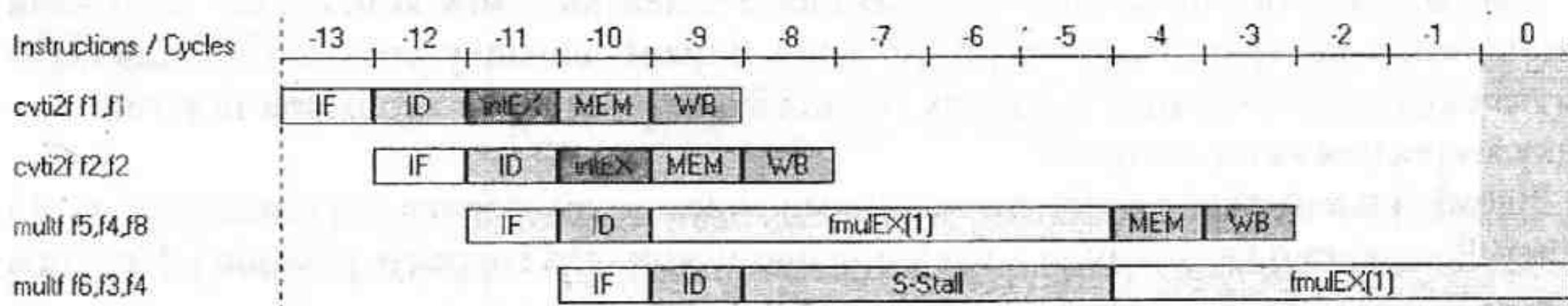


Рис. 5.2. Приклад структурного конфлікту при зверненні до пристрою множення

Структурні конфлікти другої групи виникають також при одночасному зверненні до пам'яті даних, наприклад, при необхідності читання операнда для однієї команди та запису результату для другої. Такі конфлікти виникають і в комп'ютерах з однією пам'яттю команд і даних. У цьому випадку, коли одна команда, яка знаходиться в конвеєрі, здійснює звернення до пам'яті за даними, воно конфліктуватиме з вибіркою пізнішої в черзі команди, яка вибирається з тієї ж пам'яті. В комп'ютері DLX можливий конфлікт, наприклад, при одночасному зверненні до основної пам'яті з боку кеш пам'яті даних та команд.

Зрозуміло, що комп'ютер, в якому забезпечена підтримка конвеєрного виконання команд без структурних конфліктів, завжди матиме вищу продуктивність порівняно з комп'ютером із структурними конфліктами. Виникає питання: чому розробники допускають наявність структурних конфліктів? Цьому є дві причини. По-перше, ряд структурних конфліктів принципово дуже важко або й неможливо ліквідувати для всіх випадків роботи конвеєра, наприклад, вирішити питання одночасного доступу до пам'яті

даних. По-друге, конвеєризація всіх функціональних пристрій може виявитися дуже дорогою. Комп'ютери, які допускають два звернення до пам'яті в одному такті, повинні мати пам'ять, яка характеризується подвоєною пропускною спроможністю, наприклад, це може бути досягнуто шляхом використання окремих блоків кеш пам'яті для команд і даних. Аналогічно, повністю конвеєрний пристрій ділення з рухомою комою вимагає значної кількості вентилів. Якщо структурні конфлікти не виникатимуть дуже часто, то не завжди варто платити за те, щоб їх обійти. Тому розробляється скалярний, або не повністю конвеєрний пристрій, що має меншу загальну затримку, ніж повністю конвеєрний. Наприклад, розробники пристрій з рухомою комою комп'ютерів CDC7600 і MIPS R2010 вважали за краще мати меншу затримку виконання операцій замість повної їх конвеєризації.

5.2. Конфлікти за даними

5.2.1. Типи конфліктів за даними

Одним з чинників, який істотно впливає на продуктивність конвеєрних комп'ютерів, є логічні залежності між командами. Такі залежності великою мірою обмежують потенційний паралелізм суміжних операцій виконання команд, що забезпечується відповідними апаратними засобами. Ступінь впливу цих залежностей визначається як структурою комп'ютера (в основному, організацією керування конвеєром команд і характеристиками функціональних пристрій), так і характеристиками програм.

Конфлікт за даними виникає при наявності залежності між командами, коли вони розташовані в програмі близько одна до одної. В цьому випадку для забезпечення перекриття виконання операцій залежних команд може виникнути необхідність у зміні порядку звернення за операндами.

Відомі три можливі конфлікти за даними залежно від порядку операцій читання і запису. Розглянемо дві команди – і та j , при цьому команда i передує команді j . Можливі наступні конфлікти:

1. WAR (write after read) – читання після запису. Команда j намагається прочитати ще не оновлений командою i операнд (рис. 5.3). Якби не було конвеєра, то спочатку попередня команда i записала б до комірки X операнд, який пізніше був би зчитаний командою j . У випадку використання конвеєрного опрацювання команд, як це видно з рисунка, фаза читання команди j виконується раніше фази запису команди i . Таким чином, команда j може некоректно набути старого значення.

Фази виконання команди i					Запис до комірки X
Фази виконання команди j			Читання з комірки X		

Рис. 5.3. Конфлікт “запис після читання”

2. RAW (read after write) – запис після читання. Команда j намагається записати операнд до реєстра призначення ще до того, як попередній вміст цього реєстра прочитає команда i (рис. 5.4). Якби не було конвеєра, то спочатку попередня команда i прочитала

б з комірки X операнд, а пізніше до цієї комірки був би записаний інший операнд командою j. У випадку використання конвеєрного опрацювання команд, як це видно з рисунка, фаза запису команди j виконується раніше фази читання команди i. Таким чином, команда i може набути некоректного нового значення.

Фази виконання команди i			Читання з комірки X		
Фази виконання команди j		Запис до комірки X			

Рис. 5.4. Конфлікт “читання після запису”

3. WAW (write after write) – запис після запису. Команда j намагається записати операнд до реєстра призначення ще до того, як цей запис провела команда i, тобто записи закінчуються в неправильному порядку (рис. 5.5). Якби не було конвеєра, то спочатку попередня команда i записала б до комірки X операнд, а пізніше до цієї комірки був би записаний інший операнд командою j. У випадку використання конвеєрного опрацювання команд, як це видно з рисунка, фаза запису команди j виконується раніше фази запису команди i. В результаті комірка тимчасово отримує некоректний вміст, чим може «скористатися» проміжна k-та команда.

Фази виконання команди i					Запис до комірки X
Фази виконання команди j		Запис до комірки X			

Рис. 5.5. Конфлікт “запис після запису”

Можливий також випадок RAR (read after read) – читання після читання, але він не-безпеки не створює, і тому не розглядається. Означення, класифікацію та перші методи скасування конфліктів за даними (в оригіналі – data hazards) запропонував Роберт Келлер в 1975 році.

5.2.2. Методи зменшення впливу конфліктів за даними на роботу конвеєра команд

Застосовуються наступні методи зменшення впливу зазначених вище залежностей між даними на роботу конвеєра команд:

- Призупинення виконання команди, тобто затримка з переходом бід виконання операції декодування ID до виконання операції виконання EX в конвеєрі доти, доки залежність даних не вичерпується плином часу.
- Випереджувальне пересилання з ярусів конвеєра результатів попередньої команди до потрібного яруса конвеєра, в якому виконується наступна команда (це потребує додаткових затрат обладнання та ускладнює керування).
- Статична диспетчеризація послідовності команд у програмі під час компіляції з метою зменшення впливу конфліктів за даними на роботу конвеєра команд шляхом зміни порядку виконання залежних одна від одної команд.

- Динамічна диспетчеризація послідовності команд у програмі під час компіляції з тією ж, що й статична диспетчеризація, метою.
- Перейменування реєстрів.

Розглянемо далі названі методи зменшення впливу конфліктів за даними на роботу конвеєра команд детальніше.

5.2.3. Призупинення виконання команди

Найпростішим рішенням для зменшення впливу конфліктів на роботу конвеєра команд при наявності залежності між даними є призупинення виконання команди j на декілька тактів з тим, щоб завершилось виконання команди i , або тієї її фази, яка викликала конфлікт. Відповідно затримається і виконання команд, які йдуть слідом за командою i .

Розглянемо фрагмент програми, виконання якого вимагає призупинення роботи конвеєра:

```
LW      R1, 0(R2)
SUB    R4, R1, R5
AND    R6, R1, R7
OR     R8, R1, R9
```

Рис. 5.6 ілюструє роботу конвеєра при виконанні поданого вище фрагмента програми.

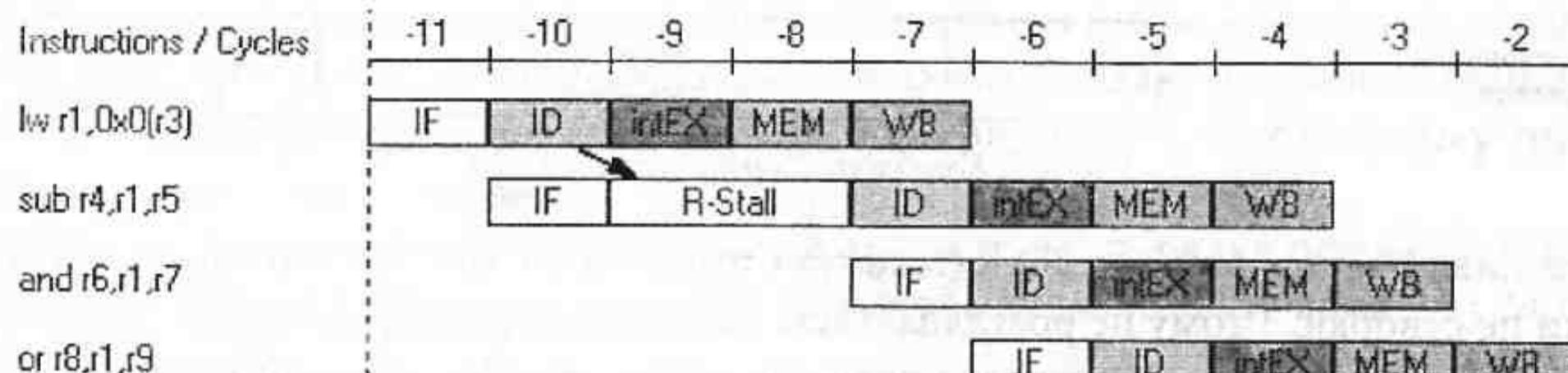


Рис. 5.6. Призупинення конвеєра

Тут друга команда SUB потребує результату виконання першої команди, який з'являється лише на такті 12 (фаза MEM), через що виконання призупиняється (зупинка R-Stall) до завершення вказаного такту. Призупинення конвеєра знижує його ефективність, тому частіше використовують інші шляхи ліквідації впливу залежностей між даними.

5.2.4. Випереджувальне пересилання

Найпоширенішим способом запобігання конфліктам в конвеєрі команд, появя яких викликана залежностями між даними, є випереджувальне пересилання даних із попередніх ярусів конвеєра в яруси, де ці дані потрібні, минаючи проміжні яруси конвеєра. Розглянемо застосування випереджувального пересилання на кількох прикладах для конкретних фрагментів коду програми.

Випереджувальне пересилання даних з метою усунення конфлікту і пригальмовання конвеєра комп'ютера DLX для коду програми

```

SUB R5, R6, R7
LW R4, 0(R5)
SW 12(R5), R6

```

ілюструє рис. 5.7, на якому показано результат виконання приведеної програми на симуляторі комп'ютера DLX WinDLX. Залежність за даними виникає за рахунок використання другою командою вмісту реєстра R5 за умови, що цей вміст оновлюється по-передньою (першою) командою фрагмента. Цю залежність долають шляхом випереджувального пересилання результату першої команди (що використовується потім як зсувний компонент адреси) з інтегрованого (до конвеєрного реєстра) поля ALUoutput (на такті -7) на вхід ALU (на такті -6). На рисунку це позначено похилою додатковою лінією. Випереджувальне пересилання ґрунтуються на тому, що реально зсувна компонента формується вчасно, вона вже існує в надрах конвеєра, але не є ще занесеною до реєстра реєстрового файла (для цього треба виконати ще фази MEM і WB).

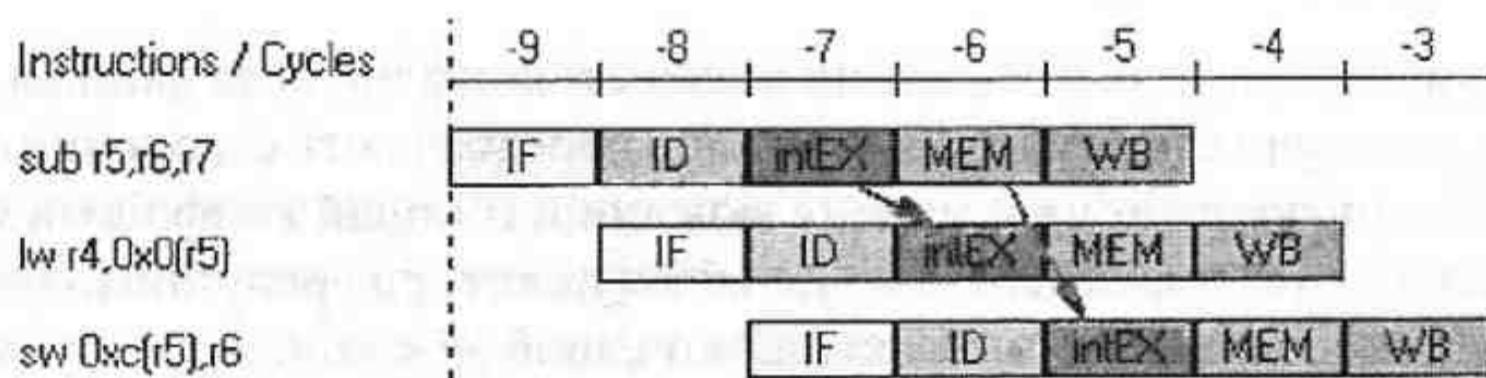


Рис. 5.7. Випереджувальне пересилання

Випереджувального пересилання вимагає і остання (третя) команда через те, що використовує вміст реєстра R4, який формується другою командою. Третій команді потрібно ще одне випереджувальне пересилання зсувної компоненти виконавчої адреси (вмісту реєстра R5), що позначено на рисунку іще однією додатковою лінією.

Потрібно зазначити, що випереджувальне пересилання не завжди є можливим. Розглянемо виконання в конвеєрі наступного фрагмента коду

```

LW R1, 0(R2)
ADD R4, R1, R2
AND R6, R1, R7
XOR R3, R1, R5.

```

Залежність даних обумовлена використанням командами, які надходять після першої команди, вмісту реєстра R1, який у конвеєрі формується із стандартною затримкою. При цьому друга команда використовує випереджувальне пересилання даних від першої команди після завершення фази MEM, коли вже отримано майбутній вміст реєстра R1. Але оскільки цей вміст отримано лише на такті -6, а він був потрібний на такті -7, то з'явилася затримка R-Stall, тому, що принципово неможливе пересилання до минулого (рис. 5.8).

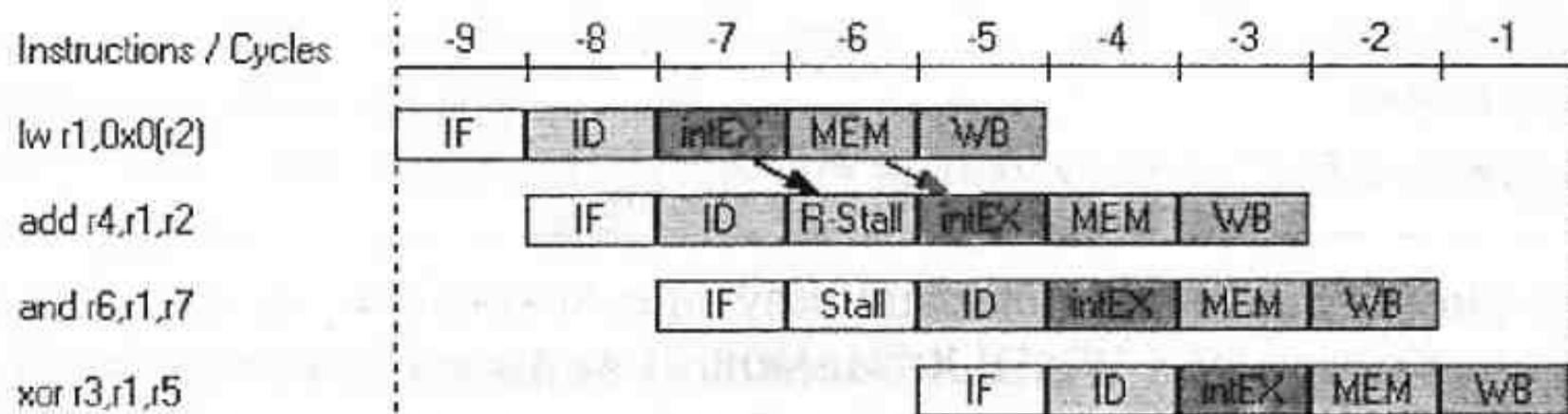


Рис. 5.8. Можливі та неможливі випереджувальні пересилання

Для забезпечення реалізації в конвеєрі випереджувального пересилання в ньому формуються додаткові канали пересилання даних з відповідними мультиплексорами для підключення входів функціональних блоків комп’ютера до цих каналів та додаткові буферні регистри

5.2.5. Статична диспетчеризація послідовності команд у програмі під час компіляції

У деяких комп’ютерах для зменшення кількості конфліктів за даними, аж до повної їх ліквідації, застосовують програмні методи, що передбачають створення оптимізуючих компіляторів, які орієнтовані на усунення можливості появи конфліктів у конвеєрі іще на стадії компіляції програми. Оптимізуючи компілятори, реорганізовують послідовність команд та створюють такий об’єктний код, щоб між конфліктуючими командами (командами, здатними до конфлікту) знаходилась достатня кількість неконфліктуючих команд. Якщо компілятору цього зробити не вдається, то він вставляє між конфліктуючими командами необхідну кількість команд типу «немає операції». Такий підхід виключає необхідність призупинення роботи конвеєра і виконання випереджувального пересилання, яке вимагає додаткових затрат обладнання та ускладнює керування

Для виявлення конфліктуючих команд в оптимізуючому компіляторі потрібно реалізувати відповідні засоби аналізу команд. Ознаками наявності конфліктів за даними є наступні

- для конфліктів за даними типу RAW – наявність збіжностей в адресах читання попередніх команд та адресах запису наступних команд,
- для конфліктів за даними типу WAR – наявність збіжностей в адресах запису попередніх команд та адресах читання наступних команд,
- для конфліктів за даними типу WAW – наявність збіжностей в адресах запису попередніх команд та адресах запису наступних команд

Тобто оптимізуючий компілятор повинен проводити аналіз на збіжність адрес запису і читання сусідніх команд, знаходити конфліктуючі команди, та розводити їх в часі

Статична диспетчеризація послідовності команд у програмі під час компіляції використовувалася, починаючи з 60-х років минулого століття, і викликала підвищений інтерес у 80-х роках, коли конвеєрні машини стали поширенішими

Нехай, наприклад, є послідовність операторів: $a = b + c$; $d = e - f$. Нижче наведена не оптимізована програма для виконання цих операторів

LW r2,0(r0)
LW r3,4(r0)
ADD r4,r2,r3

SW 8(r0),r4
 LW r5,12(r0)
 LW r6,16(r0)
 SUB r7,r5,r6
 SW 20(r0),r7

Вхідні дані та результати цієї програми розміщені в комірках пам'яті даних, наведені в табл. 5.1.

Таблиця 5.1

Операнд	Адреса комірки пам'яті
a	8
b	0
c	4
d	20
e	12
f	16

Діаграма виконання програми приведена на рис. 5.9. Як видно з рисунка, тут наявні чотири затримки, викликані залежністю за даними: друга і третя команди – звернення до регістра r3, третя і четверта команди – звернення до регістра r4, шоста і сьома команди – звернення до регістра r6, сьома і восьма команди – звернення до регістра r7.

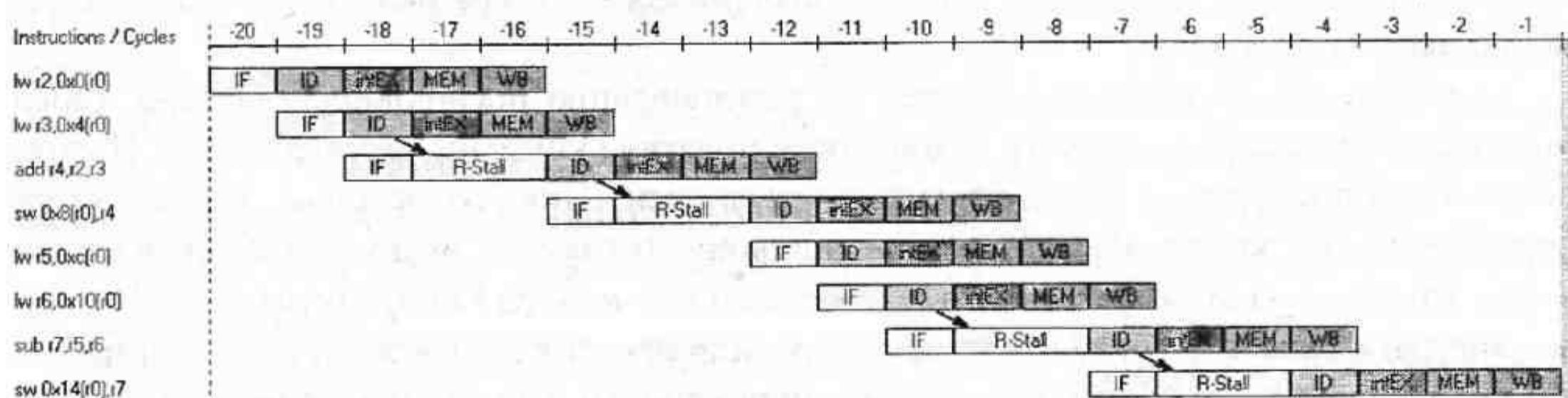


Рис. 5.9. Діаграма виконання неоптимізованої програми

Знаходимо конфліктуючі команди в наведеній вище програмі та розводемо їх в часі. Нижче наведена оптимізована програма для виконання тих же операторів

LW r2,0(r0)
 LW r3,4(r0)
 LW r5,12(r0)
 LW r6,16(r0)

ADD r4,r2,r3
 SUB r7,r5,r6
 SW 8(r0),r4
 SW 20(r0),r7

Вхідні дані та результати розміщені в тих же комірках пам'яті, приведених в табл. 5.1. Тоді діаграма виконання програми буде мати вигляд, як це приведено на рис. 5.10 (отримана з використанням симулятора WinDLX).

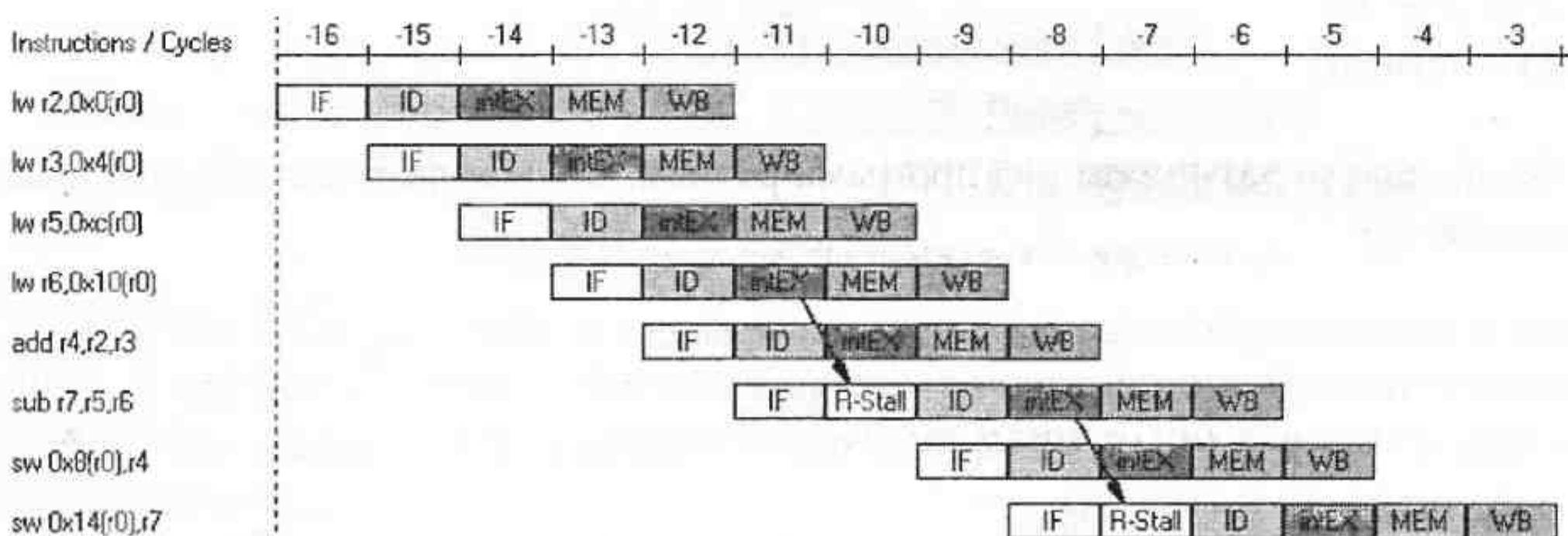


Рис. 5.10. Діаграма виконання оптимізованої програми

Як наслідок, повністю усунені два блокування (стосовно регістрів r3 і r4), та в двох блокуваннях (стосовно r6 і r7) забрано по одному такту затримки. Тобто в цілому забрано 6 тактів. Залежності, які залишилися, можна забрати, використавши схеми обходу.

Відзначимо, що використання різних регістрів для першого і другого операторів було достатньо важливим для реалізації такого правильного планування. Зокрема, якби змінна e була б завантажена в той же самий регістр, що b або c, то таке планування не було б коректним. У загальному випадку планування конвеєра може вимагати збільшеної кількості регістрів.

Статична диспетчеризація передбачає реорганізацію послідовності команд, серед яких відсутні команди переходу, за винятком початку і кінця цієї послідовності. Планування такої послідовності команд здійснюється досить просто, оскільки в компіляторі передбачено, що кожна наступна команда виконуватиметься, якщо виконується перша з них. Тому можна побудувати граф залежностей цих команд і впорядкувати їх так, щоб мінімізувати кількість призупинень конвеєра. Для простих конвеєрів використання статичної диспетчеризації дає цілком задовільні результати. Проте, коли довжина конвеєра збільшується і його затримки зростають, потрібні складніші алгоритми диспетчеризації.

5.2.6. Динамічна диспетчеризація послідовності команд у програмі під час компіляції

Динамічна диспетчеризація передбачає проведення компіляції програми узгоджено з процесом виконання команд в комп'ютері з метою мінімізації кількості призупинень конвеєра. У англомовній літературі разом з терміном "динамічна диспетчеризація" часто застосовуються терміни "out-of-order execution" – невпорядковане виконання і "out-of-order issue" – невпорядковане видавання, які практично завжди наявні при динамічній диспетчеризації. При динамічній оптимізації в конвеєрі команд розміщується пристрій виявлення конфліктів, який інформує компілятора про наявність конфліктів з тим, щоб він міг реагувати в процесі компіляції на ці конфлікти. Одним з варіантів реагування є

буферизація команд, що чекають вирішення конфлікту, і подання подальших, логічно не пов'язаних команд, в конвеєр. При цьому буферизовані команди можуть подаватися в потрібний ярус конвеєра, що забезпечується створенням в ньому комутуючих магістралей, які, крім того, забезпечують засилання результату операції безпосередньо в буфер, що зберігає логічно залежну команду, затриману через конфлікт, або безпосередньо на вход функціонального пристрою до того, як цей результат буде записаний в реєстровий файл або в пам'ять.

При динамічній диспетчеризації команди можуть подаватися на виконання не в тому порядку, в якому вони розташовані в програмі, проте засоби виявлення і усунення конфліктів між логічно зв'язаними командами мають забезпечувати отримання результатів відповідно до заданої програми.

5.2.7. Перейменування реєстрів

Ще одним апаратним методом мінімізації конфліктів за даними є метод перейменування реєстрів. При реалізації цього методу в адресних полях команди вказуються номери не фізичних, а логічних (увівих) реєстрів. Номери логічних реєстрів динамічно відображаються на номери фізичних реєстрів, які розміщаються в реєстровому файлі процесора, за допомогою таблиць відображення, котрі оновлюються після декодування кожної команди. Кожен новий результат записується в фізичний реєстр.

Проте попереднє (тимчасове) значення кожного логічного реєстра зберігається і може бути відновлене, якщо виконання команди має бути перерване через виникнення виняткової ситуації або у випадку невірного передбачення напряму умовного переходу. В процесі виконання програми генерується велика кількість тимчасових результатів. Ці тимчасові результати записуються в реєстрові файли разом з постійними результатами. Тимчасовий результат стає новим постійним результатом, коли завершується виконання команди. У свою чергу, завершення виконання команди відбувається, коли виконання всіх попередніх команд успішно завершено в заданому програмою порядку.

Програміст має справу тільки з логічними реєстрами. Реалізація фізичних реєстрів від нього прихована.

Метод перейменування реєстрів спрощує контроль залежностей за даними. У комп'ютері, який може виконувати команди не у порядку їх розташування в програмі, номери логічних реєстрів можуть стати двозначними, оскільки один і той же реєстр може бути призначений послідовно для зберігання різних значень. Але оскільки номери фізичних реєстрів унікально ідентифікують кожен результат, всі неоднозначності усуваються.

5.3. Конфлікти керування

5.3.1. Типи конфліктів керування

Конфлікти керування можуть викликати ще більше зниження продуктивності конвеєра, ніж конфлікти за даними. Вони викликані наявністю в програмах команд керування, які змінюють хід обчислювального процесу: безумовний та умовний переходи, пропуск, виклик процедури та повернення з неї. Виконання команд керування може

призводити до необхідності очистки конвеєра та завантаження нової послідовності команд, що знижує його продуктивність. Те, що команда належить до команд керування, можна вияснити лише після її декодування, тобто за кілька тактів після її надходження до конвеєра (в комп'ютері DLX через 2 такти).

За цей час на перші яруси конвеєра вже надійдуть нові команди. При виявленні команди керування потрібно вияснити, чи здійснюється переход, і якщо так, то очистити конвеєр від наступних за нею команд та завантажити команду, розміщену за адресою переходу. Для цього спочатку потрібно визначити виконавчу адресу переходу, яка може бути або безпосередньо в адресному полі команди, або її потрібно обчислити в наступних ярусах конвеєра. Таким чином, реалізація в конвеєрі команд керування вимагає виконання додаткових операцій, що рівнозначно його зупинці на відповідну кількість тактів.

Особливо складною для виконання в конвеєрі є команда умовного переходу. Коли виконується команда умовного переходу, вона може або змінити вміст лічильника команд, або залишити його без змін. Якщо команда умовного переходу замінює лічильник команд значенням адреси, обчисленої в команді, то переход називається здійсненим. В іншому випадку він називається нездійсненим. Для забезпечення опрацювання в конвеєрі команд умовного переходу потрібно передбачити проведення відповідних дій. Простий метод роботи з умовними переходами полягає в призупиненні конвеєра як тільки виявлена команда умовного переходу до тих пір, поки вона не досягне яруса конвеєра, який обчислює нове значення лічильника команд. Реалізація цього методу для наведеного нижче фрагмента програми

```
dd r7,r8,r1
bnez r4,$TEXT
and r5,r7,r7
add r2,r2,r3
```

відображена на рис. 5.11, де після декодування команди BNEZ (переход, якщо не рівний нулю) призупиняється виконання наступної команди.

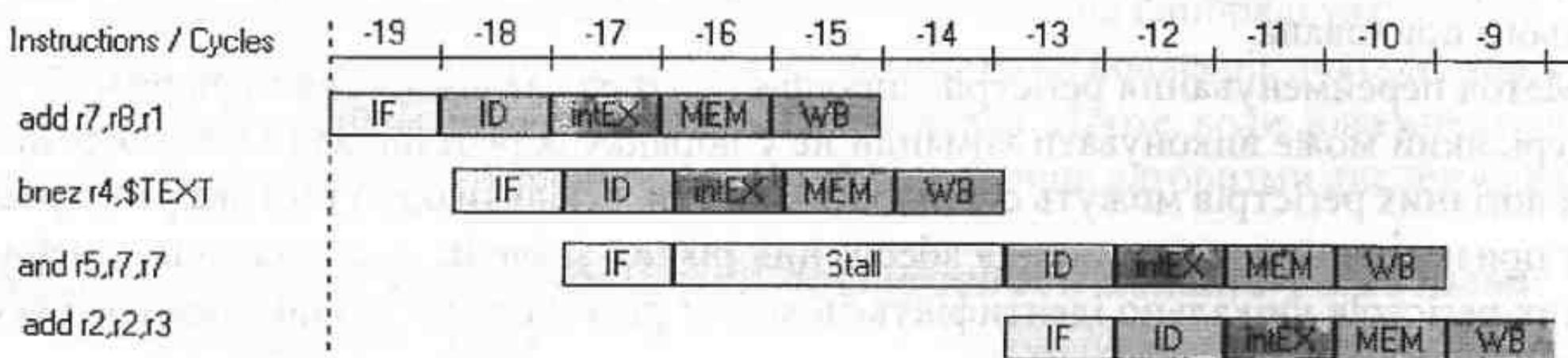


Рис. 5.11. Призупинення конвеєра при виконанні команди умовного переходу

Комп'ютер, в якому здійснюються призупинення при виявленні команд умовних переходів, суттєво втрачає прискорення, що одержується за рахунок конвеєрної організації. При цьому, чим більша глибина конвеєра, тим більші втрати на командах умовного переходу.

Таким чином, для зменшення втрат в конвеєрі через конфлікти керування потрібно звернути увагу в першу чергу на організацію вибірки команди, до якої здійснюється переход, та на скорочення часу виконання команд умовного переходу.

5.3.2. Зниження втрат на вибірку команди, до якої здійснюється перехід

Для зниження втрат на вибірку команди, до якої здійснюється перехід, використовуються декілька підходів:

- обчислення виконавчої адреси команди переходу в ярусі декодування команди;
- використання буфера адрес переходів;
- використання буфера команд, до яких здійснюються переходи;
- використання буфера циклу.

Обчислення виконавчої адреси команди переходу в ярусі декодування команди передбачає на етапі декодування команди визначення не тільки її належності до команд керування, але і адреси переходу. Ця адреса може бути в полі самої команди, а якщо ні, то має бути обчислена в цьому ж ярусі. Для забезпечення обчислення адреси переходу в ярусі декодування вводиться додатковий суматор. Тим самим час зупинки конвеєра може бути зменшений до одного такту.

При використанні буфера адрес переходів, який є асоціативною пам'яттю, в ньому зберігаються адреси декількох останніх команд переходів, які є ознакою пошуку, та відповідні їм адреси переходу. Перед вибіркою з пам'яті чергової команди її адреса (вміст лічильника команд) порівнюється з адресами наявних в буфері команд і, якщо відбулося співпадіння, з буфера вибирається адреса переходу (рис. 5.12), а сигнал про наявність адреси в буфері пропускає її через мультиплексор. Тоді вибірка команди переходу може бути здійснена вже в наступному такті. При відсутності адреси команди в буфері адреса переходу шукається в ярусі декодування та, разом з адресою команди, заноситься до буфера адрес переходів. При необхідності заміщення інформації в буфері адрес переходів використовуються алгоритми заміщення типу FIFO, LRU, RAND.



Рис. 5.12. Використання буфера адрес переходів

Кращим, ніж використання буфера адрес переходів, є використання буфера команд, до яких здійснюється переход (рис. 5.13), коли в буфер разом з адресою команди переходу, яка є ознакою пошуку, записуються коди команд, до яких здійснюються переходи, що дозволяє при повторному виконанні такої команди виключити не тільки фазу обчислення адреси переходу, але і фазу вибірки команди. При відсутності в буфері команди, до якої здійснюється переход, її адреса визначається в ярусі декодування, проводиться її зчитування з пам'яті команд та, разом з адресою команди переходу, вона заноситься до буфера.



Рис. 5.13. Використання буфера команд переходів

Особливо ефективним є використання буфера адрес переходів та буфера команд, до яких здійснюється переход при виконанні циклів.

Іще більший ефект дає використання буфера циклу, до якого із збереженням попереднього порядку записується деяка кількість команд, що виконувались останніми. Буфер входить до першого ярусу конвеєра, в якому виконується вибірка команд (рис. 5.14). Коли відбувається переход, то спочатку здійснюється звернення до буфера. Якщо це був повторний цикл або ітерація, то всі його команди будуть вже наявними в буфері, який є меншим за об'ємом і швидшим основної пам'яті та кеш пам'яті команд. Тим самим всі наступні ітерації будуть виконуватись із зчитуванням команд з буфера, що суттєво прискорює роботу комп'ютера. Описаний підхід зокрема був використаний в комп'ютерах CRAY-1 фірми Cray Research та CDC 6600 фірми Control Data Corporation.



Рис. 5.14. Використання буфера циклу

5.3.3. Зниження втрат на виконання команд умовного переходу

Число тактів, що втрачаються під час призупинення конвеєра через наявність умовних переходів, може бути зменшено наступними способами:

- Введенням буфера попередньої вибірки для виявлення, здійснимим чи нездійснимим є умовний переход та підготовки до переходу на початкових ярусах конвеєра.
- Дублюванням початкових ярусів конвеєра для підготовки до переходу, аж до виявлення, здійснимим чи нездійснимим є умовний переход.
- Затримкою переходу, тобто виконанням наступних за командою переходу команд незалежно від напряму переходу.
- Статичним та динамічним передбаченнями переходу, тобто попереднім обчисленням значення лічильника команд (цільової адреси) для здійсненного переходу.

5.3.3.1. Введення буфера попередньої вибірки

Буфер попередньої вибірки встановлюється після яруса вибірки команди. Він складається з двох блоків пам'яті типу FIFO та блоку обчислення цільової адреси переходу, і включений в конвеєр команд, як це показано на рис. 5.15.

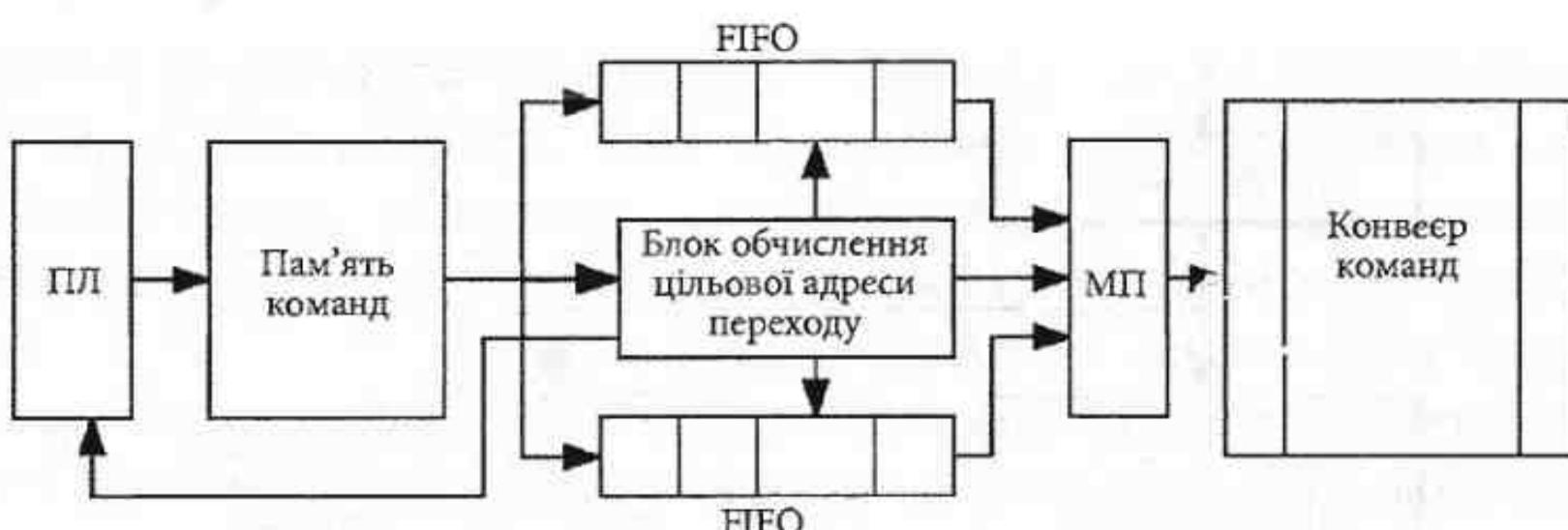


Рис. 5.15. Конвеєр з буфером попередньої вибірки команд

За значенням програмного лічильника ПЛ команди з пам'яті команд зчитуються в один з буферів FIFO. Блок обчислення цільової адреси аналізує кожну зчитану команду. При виявленні команди умовного переходу він обчислює адресу переходу та керує блоками FIFO і програмним лічильником з тим, щоб забезпечити паралельне зчитування обох можливих послідовностей команд з адреси переходу в блоки FIFO. Після обчислення адреси переходу блок обчислення цільової адреси підключає до входу конвеєра команд через мультиплексор МП відповідний буфер, а вміст іншого буфера стирається.

За рахунок попередньої вибірки для виявлення того, здійсненим чи нездійсненим є умовний переход, таким підходом вдається зменшити втрати на виконання команд умовного переходу. При цьому буфер попередньої вибірки можна розглядати як декілька додаткових ярусів конвеєра, які збільшують його початкову затримку та не впливають на продуктивність. З рис. 5.12 видно, що такий підхід ускладнює роботу конвеєра. Крім того, при наявності в програмі декількох команд умовного переходу підряд це вимагає включення додаткових блоків FIFO та іще більше ускладнює роботу комп'ютера.

5.3.3.2. Дублювання початкових ярусів конвеєра

Подібним до описаного вище способом зниження втрат на виконання команд умовного переходу є дублювання початкових ярусів конвеєра, тобто створення двох паралельних гілок початкових ярусів конвеєра команд, як це показано на рис. 5.16.

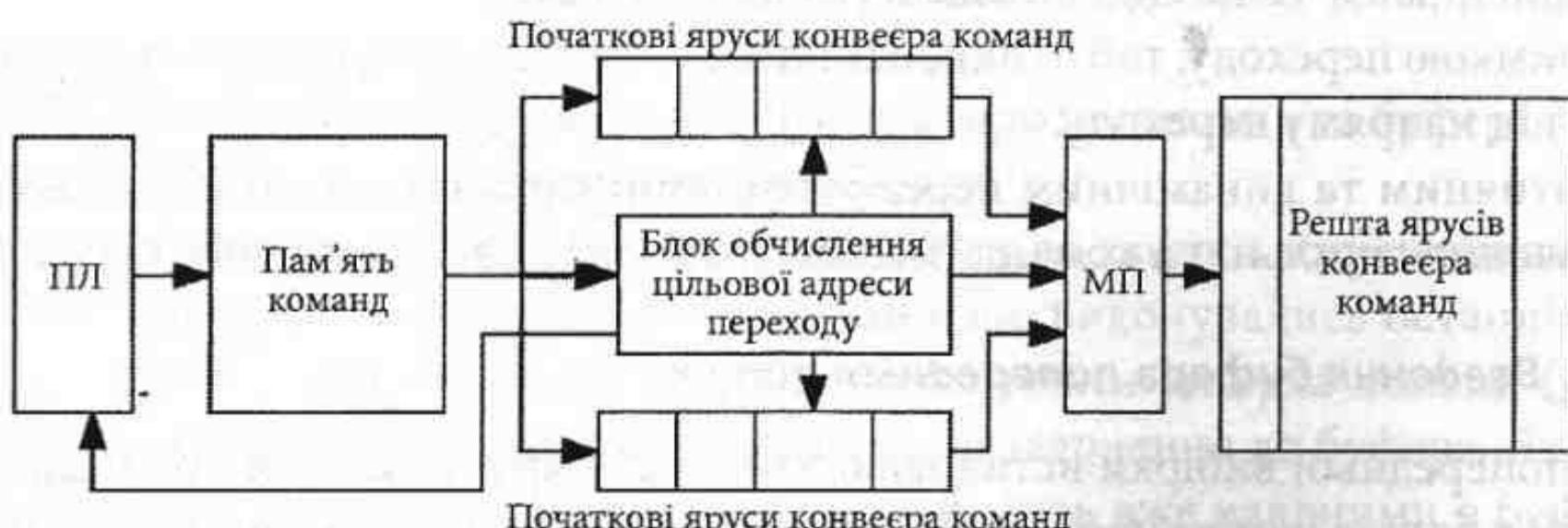


Рис. 5.16. Конвеєр з дублюванням початкових ярусів

В одній із паралельних гілок початкових ярусів конвеєра команд послідовність команд відповідає випадку, коли умова переходу виконується, а в іншій – коли не виконується. Кількість ярусів у цих вітках визначається кількістю тактів, потрібних для обчислення адреси переходу блоком обчислення цільової адреси переходу. Після отримання адреси переходу блок обчислення цільової адреси переходу через мультиплексор МП

підключає до решти ярусів конвеєра команд відповідну вітку початкових ярусів.

Тут також з'являються проблеми, коли до прийняття рішення щодо поточної команди переходу в конвеєр надходить нова команда. Тоді знову вимагаються додаткові паралельні вітки початкових ярусів конвеєра команд.

Описаний метод, як і попередній, знайшов застосування в кількох версіях сім'ї комп'ютерів IBM 360/370.

5.3.3.3. Затримка переходу

Ми вже бачили, що проста схема виконання в конвеєрі команд умовного переходу полягає в блокуванні виконання будь-якої команди, наступної за командою умовного переходу, до тих пір, поки не стане відомим напрям переходу. Рис. 5.11 відображав саме такий підхід. Привабливість такого рішення полягає в його простоті.

На противагу цьому методу стратегія затриманого переходу передбачає продовження виконання команд, які ідуть в програмі за командою переходу, незалежно від її результату. Це має сенс лише тоді, коли наступні за командою переходу команди мають бути виконаними незалежно від того, був переход чи ні, а команда переходу не має на них впливу.

Пошук таких команд здійснюється на етапі компіляції програми, а якщо знайдена їх кількість недостатня, то на вільні місця вставляються команди типу «немає операції».

5.3.3.4. Статичне передбачення переходу

Передбачення переходу є одним з найефективніших способів подолання конфліктів керування. Ідея цього способу полягає в тому, що до моменту виконання команди умовного переходу, або відразу після її надходження в конвеєр робиться припущення про напрям виконання умовного переходу залежно від імовірності його здійсненості або нездійсненості. Команди подаються в конвеєр відповідно до прийнятого припущення. Якщо припущення виявилося правильним, то жодних втрат, пов'язаних із командою переходу, в конвеєрі не буде. При помилковому припущення конвеєр необхідно повернути до стану, з якого почалася помилкова вибірка команд, що рівнозначно призупиненню конвеєра на втрачену кількість тактів. Вигода від застосування цього методу тим більша, чим вища точність передбачення, тобто відношення кількості правильних передбачень до загальної їх кількості.

На даний час розроблено багато способів передбачення переходу. Залежно від того, на основі чого робиться передбачення, розрізняють статичне та динамічне передбачення переходів.

Статичне передбачення переходів здійснюється на етапі компіляції програми на основі деякої априорної інформації про неї. Найшире застосування знайшли наступні методи статичного передбачення умовного переходу:

- метод повернення;
- метод профілювання, передбачення здійснюється за результатами використання інформації про профіль виконання програми, яка зібрана в результаті попередніх її запусків;
- результат переходу визначається кодом операції команди переходу;
- результат переходу визначається напрямом переходу.

Метод повернення ґрунтуються на передбаченні, що переход не відбувається ніколи,

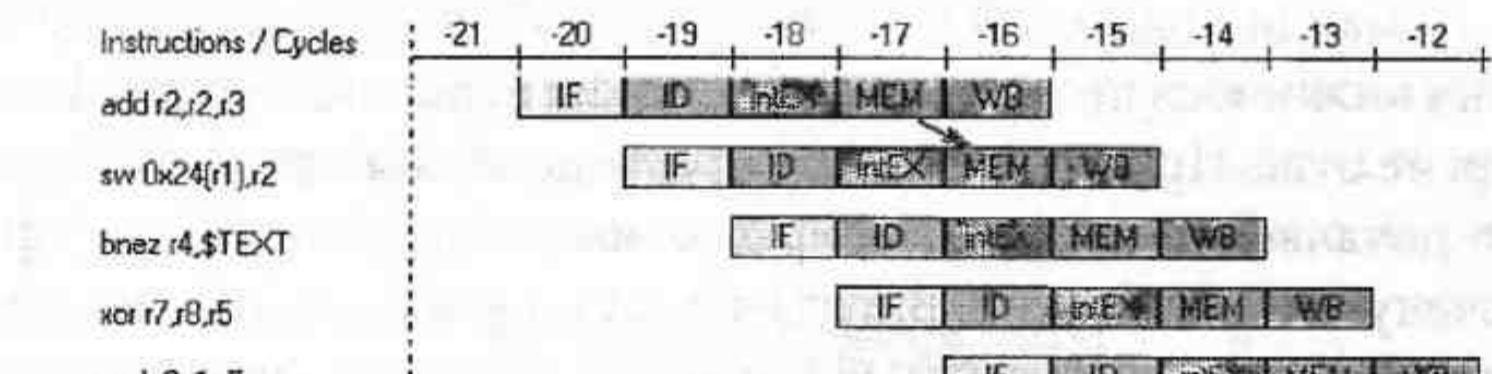
або що перехід відбувається завжди. В першому випадку умовний перехід прогнозується як нездійснений. При цьому апаратура повинна просто продовжувати виконання програми, неначебто умовний перехід зовсім не виконувався. В цьому випадку необхідно поклопотатися про те, щоб не змінити стан комп'ютера до тих пір, поки напрям переходу не стане остаточно відомим.

У деяких комп'ютерах ця схема з нездійсненими за прогнозом умовними переходами реалізована шляхом продовження вибірки команд, неначе умовний перехід був звичайною командою. Поведінка конвеєра виглядає так, ніби нічого незвичайного не відбувається (рис. 5.17a). Проте, якщо умовний перехід насправді виконується, то необхідно просто очистити конвеєр від команд, вибраних слідом за командою умовного переходу, і заново повторити вибірку команд (рис. 5.17b).

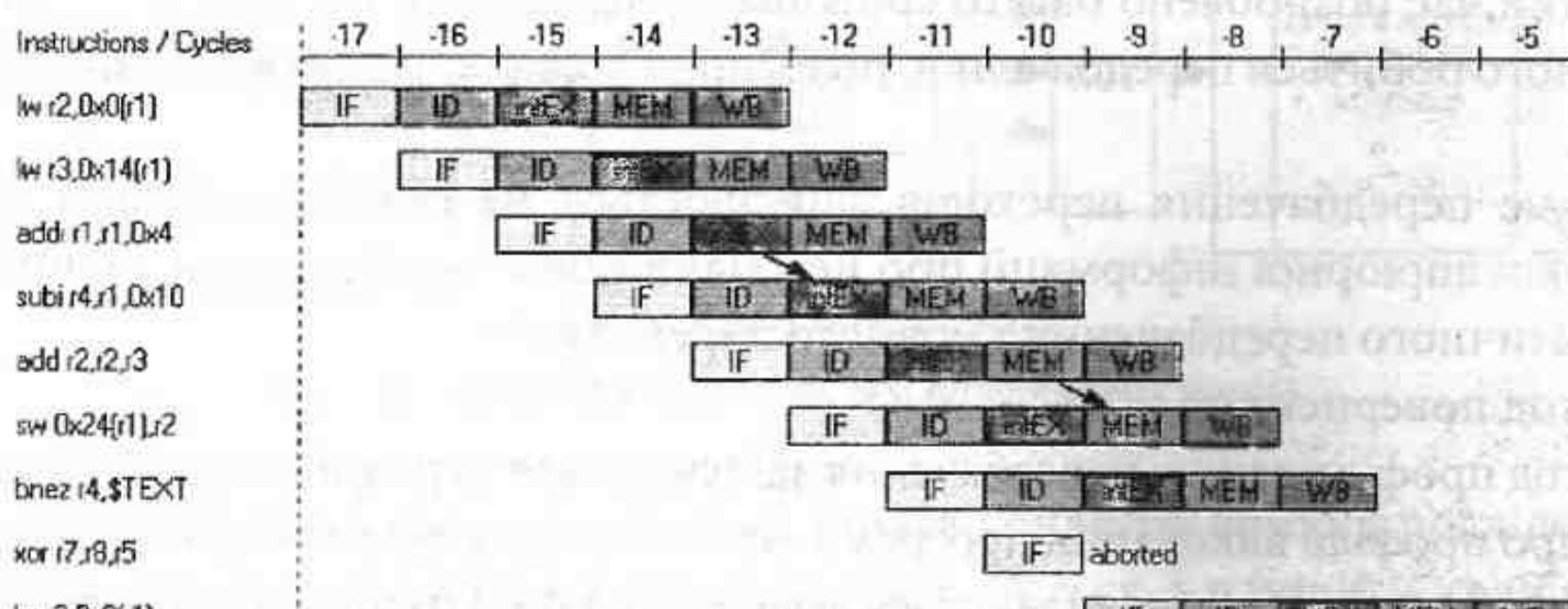
Розглянемо фрагмент програми

loop:

```
lw r2,0(r1)
lw r3,20(r1)
addi r1,r1,#4
subi r4,r1,#16
add r2,r2,r3
sw 36(r1),r2
bnez r4,loop
xor r7,r8,r5
and r2,r1,r5
add r3,r8,r2
trap 0
```



a)



b)

Рис. 5.17. Діаграма роботи модифікованого конвеєра

Альтернативна схема прогнозує перехід як здійснений. Як тільки команда умовного переходу декодована і обчислена цільова адреса переходу, припускається, що перехід здійснений, і проводиться вибірка команд та їх виконання, починаючи з цільової адреси.

Метод повернення є простим в реалізації і достатньо ефективним для деяких класів програм, тому широко використовується в комп'ютерах, зокрема в MIPS-X, SuperSPARC, I486, M68020, MC88000, VAX11/780.

Під профілюванням розуміється виконання програми для деякого еталонного набору вихідних даних, в процесі якого збирається інформація про результат кожної команди умовного переходу. Тобто тут прогноз переходів базується на інформації про профіль виконання програми, зібраної під час попередніх її проходжень. Ключовим моментом тут є те, що поведінка переходів при виконанні програми часто повторюється. Ті команди, які частіше закінчуються переходом, прогнозуються як здійсненні, інші – як нездійсненні. Вибір фіксується в спеціальному розряді команди. При виконанні програми конвеєр команд враховує значення цього розряду. Проведені дослідження показують достатньо успішне прогнозування переходів з використанням цієї стратегії.

Іще одним досить простим і ефективним є метод статичного передбачення умовного переходу за кодом операції команди переходу. Тут одні команди прогнозуються як здійсненні, наприклад “більше нуля”, а інші – як нездійсненні, наприклад “переповненні”

Перехід в програмі може відбутися в двох напрямках – вперед або назад, залежно від того, більша адреса переходу від вмісту програмного лічильника чи менша. Логічно передбачити, що команди з напрямком переходу назад є більш імовірними, ніж команди переходу вперед, оскільки більшість команд переходу використовується для організації циклів, коли відбуваються переходи до початку циклу. Ця стратегія і покладена в основу методу передбачення результата переходу за напрямом переходу.

5.3.3.5. Динамічне передбачення переходу

Динамічне передбачення переходу здійснюється в ході обчислень, виходячи з інформації про попередні переходи. Порівняно зі статичним динамічне передбачення має вищу точність, тобто більше припущень є правильними, але є значно складнішим.

При реалізації методів динамічного передбачення створюється таблиця історії переходів.

Найпростішим варіантом є однорозрядна таблиця історії переходів, в якій зберігається результат останнього виконання команди переходу. Якщо ця команда завершилася переходом, то у відповідну комірку таблиці записується одиниця, в іншому випадку – нуль. Передбачення переходу для чергової команди збігається із здійсненим переходом попередньої. Після виконання цієї команди, якщо передбачення не здійснилося, вміст комірки таблиці коригується.

Таблиця історії переходів реалізується в складі буфера адрес переходу (рис. 5.12). Кожен рядок буфера адрес переходу включає адресу команди переходу, прогнозовану адресу наступної команди (адресу переходу) і передісторію команди переходу (рис. 5.18). Біти передісторії є інформацією про виконання або невиконання умов переходу даної команди у минулому. Звернення до буфера адрес переходу (порівняння з полями адрес команд переходу) проводиться за допомогою поточного значення програмного лічильника на етапі вибірки чергової команди. За передісторією команди прогнозується виконання або

невиконання умов команди переходу і проводиться вибірка та дешифрування команд із прогнозованої гілки програми. При цьому, якщо виявлений збіг, то дана команда є командою умовного переходу, і адреса переходу має бути використаною в якості наступного значення програмного лічильника, якщо збігу немає, то команда не є командою переходу.

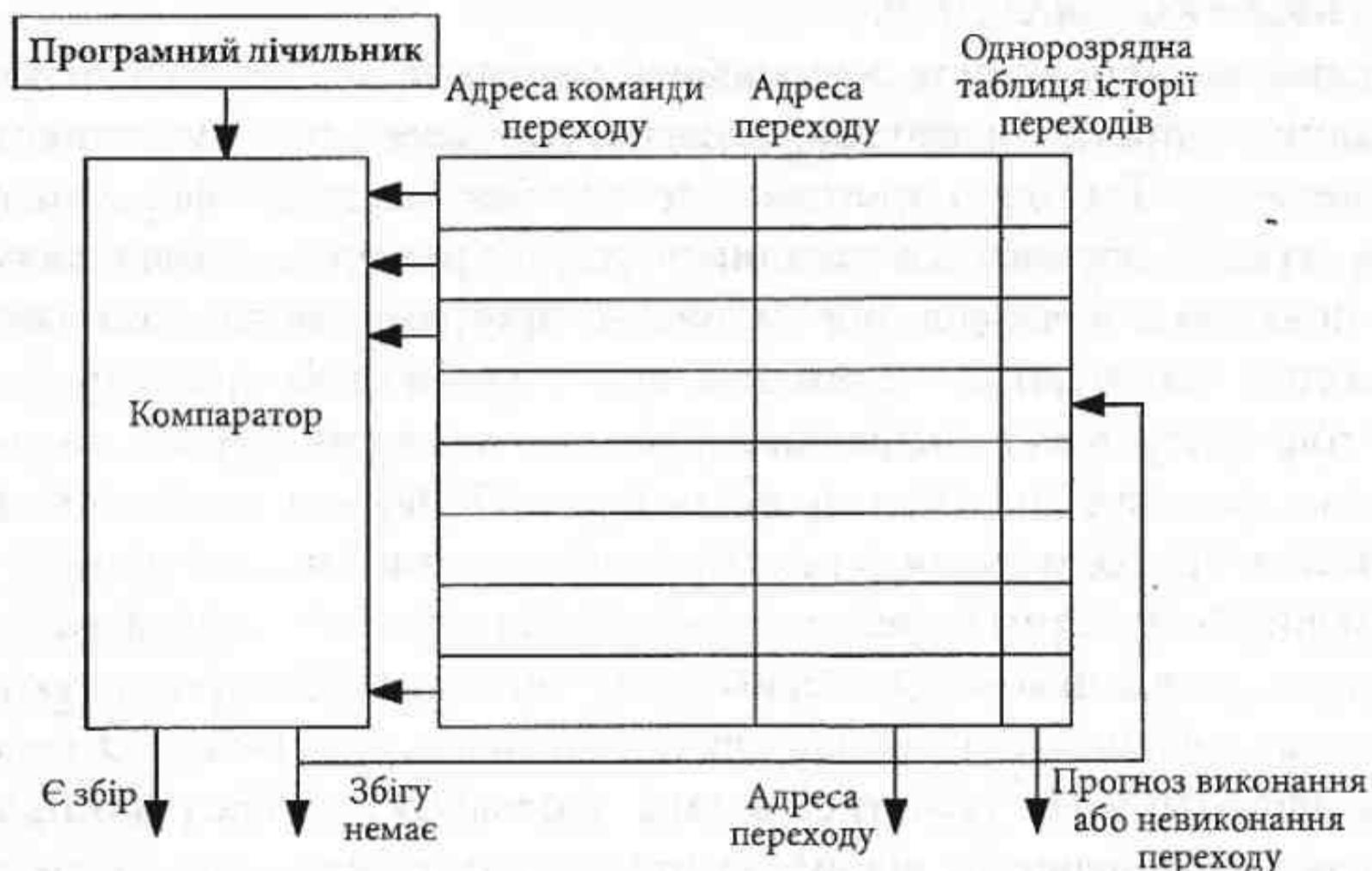


Рис. 5.18. Динамічне передбачення переходу

Більш ефективним є використання таблиці історії переходів з більшою розрядністю комірок. Практичне використання знайшли таблиці з дво- та трирозрядними комірками. Вважається, що передісторія переходу, що містить інформацію про два попередні випадки виконання цієї команди, дозволяє прогнозувати розвиток подій з цілком достатньою вірогідністю. При надходженні команди умовного переходу в конвеєр відбувається звернення до таблиці історії переходів та, залежно від вмісту відповідної комірки, робиться прогноз, який визначає подальший порядок читання команд програми. Після визначення фактичного результату переходу до вмісту комірки додається одиниця, якщо переход відбувся, та віднімається одиниця, якщо переход не відбувся.

В якості адреси таблиці історії переходів може бути використана адреса команди умовного переходу, вміст реєстру локальної історії або реєстру глобальної історії, та комбінація вказаних даних. Цим визначається вибрана стратегія динамічного передбачення.

Якщо в якості адреси таблиці історії переходів використовується адреса команди умовного переходу, тобто вміст програмного лічильника, як це показано на рис.5.18, то такий підхід дозволяє враховувати поведінку кожної команди умовного переходу, яка в більшості випадків є, як правило, здійсненою або, зазвичай, нездійсненою. Використання таблиці історії переходів дозволяє розділити команди із здійсненим і з нездійсненим умовним переходом. Функціонування цього способу формування коду передбачення, який має назву однорівневої схеми передбачення, для випадку, коли таблиця історії переходів є кількарозрядною, показано на рис. 5.19а.

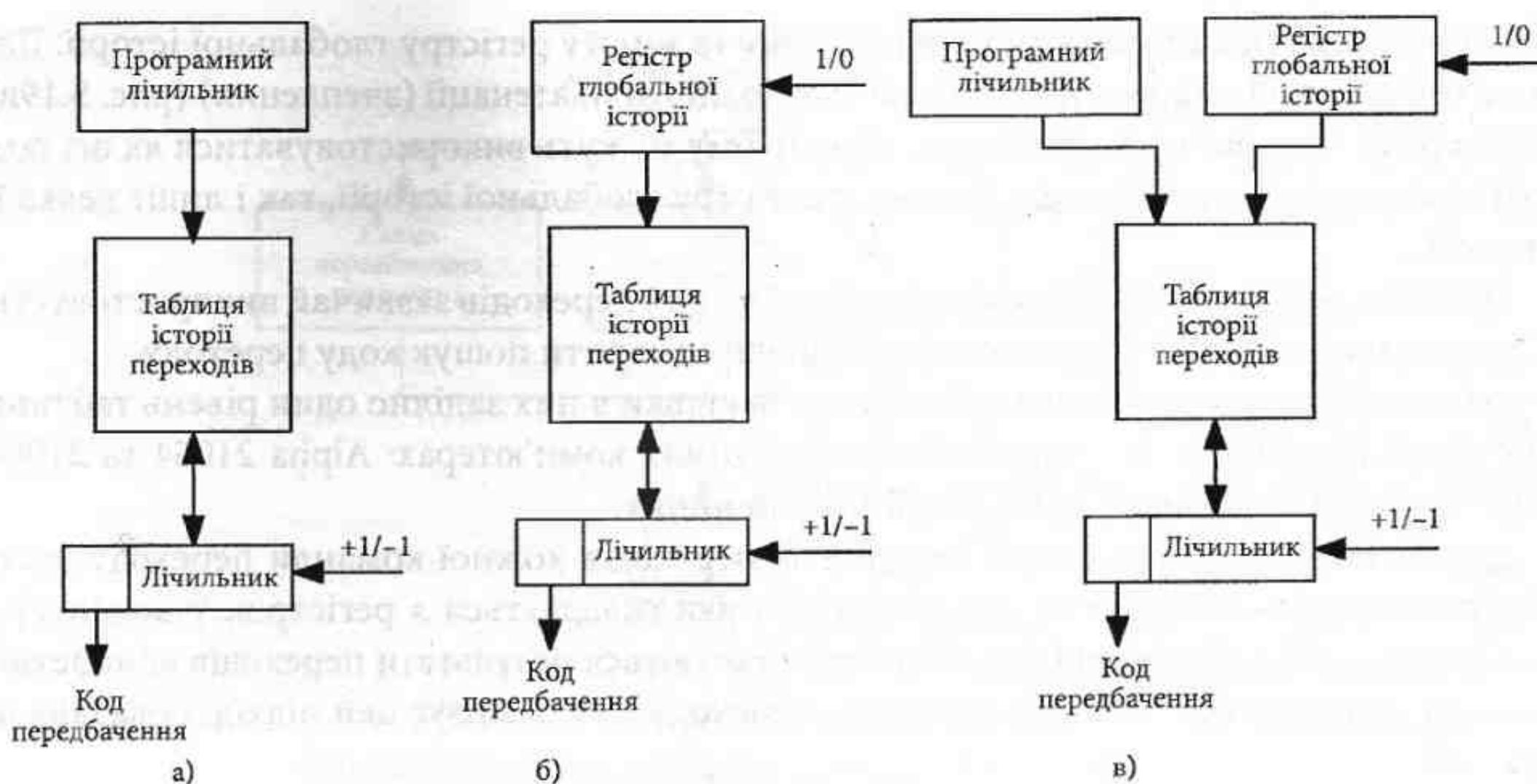


Рис. 5.19. Однорівнева схема передбачення з формуванням адреси таблиці історії переходів: а – у програмному лічильнику; б – в реєстрі глобальної історії; в – з комбінованим формуванням адреси

Вміст комірки, зчитаний з таблиці історії переходів за адресою з програмного лічильника, записується в лічильник, в якому здійснюється додавання або віднімання одиниці. Лічильник працює в режимі насищення, тобто його вміст не змінюється при додаванні одиниці, коли він має максимальне значення, та не змінюється при відніманні одиниці, коли він має нульове значення. В якості передбачення використовується старший розряд лічильника. Якщо він рівний одиниці, то передбачається, що переход є здійснений, якщо нуль – нездійснений. Значення цього розряду надходить в конвеєр для керування вибіркою подальших команд, а вміст лічильника після модифікації повертається за тією ж адресою в таблицю.

Описаний підхід забезпечує високу імовірність передбачення для багатократно виконуваних команд переходу. Однак для одноразово виконуваних команд переходу цей підхід не діє. Тому для таких команд переходу потрібно врахувати результати переходу попередніх команд, оскільки між ними є взаємозалежність, і це дозволяє підвищити кількість правильних передбачень. Для забезпечення врахування результатів переходу попередніх команд до схеми передбачення вводиться реєстр глобальної історії, вміст якого відображає історію виконання n останніх команд умовного переходу, де n – розрядність реєстра. Це є зсувний реєстр, вміст якого зсувается на один розряд після кожного виконання команди умовного переходу, а до звільненого розряду заноситься одиниця або нуль залежно від наявності чи відсутності переходу відповідно. Кожному значенню реєстра відповідає своя комірка в таблиці історії (рис. 5.19б). Вміст цієї комірки модифікується як і в попередньому способі, а її старший розряд передбачає результат команди переходу.

Підвищення точності передбачення досягається одночасним врахуванням як результатів попереднього виконання даної команди переходу, так і результатів виконання інших команд переходу. Це реалізується формуванням адреси таблиці історії переходів

шляхом об'єднання адреси команди переходу та вмісту регістру глобальної історії. Для такого об'єднання використовується або операція конкатенації (зчеплення) (рис. 5.19в), або операція додавання за модулем 2. При цьому можуть використовуватися як всі розряди адреси команди переходу та вмісту регістру глобальної історії, так і лише деяка їх кількість.

Потрібно відзначити, що в якості таблиці історії переходів зазвичай використовується асоціативна пам'ять, що дозволяє суттєво прискорити пошук коду переходу.

Наведені схеми названі однорівневими, оскільки в них задіяно один рівень таблиць. Такі схеми передбачення використані в наступних комп'ютерах: Alpha 21064 та 21064, AMD K5, R10000, Power PC620, UltraSPARC та інших.

Для врахування конкретних результатів переходівожної команди переходу часто використовується таблиця локальної історії, яка складається з регистрів, у кожному з яких подібно до регістру глобальної історії фіксуються результати переходів конкретної команди. Дворівнева схема передбачення переходу, яка реалізує цей підхід, показана на рис. 5.20.



Рис. 5.20. Дворівнева схема передбачення переходу з використанням таблиці локальної історії

Для різних програм різні стратегії передбачення дають різну точність. Тому в ряді комп'ютерів застосовуються гіbridні схеми передбачення переходів, коли в конкретному випадку застосовується та схема передбачення, від якої очікується найвища точність передбачення. Структура такої гіbridної схеми показана на рис. 5.21.

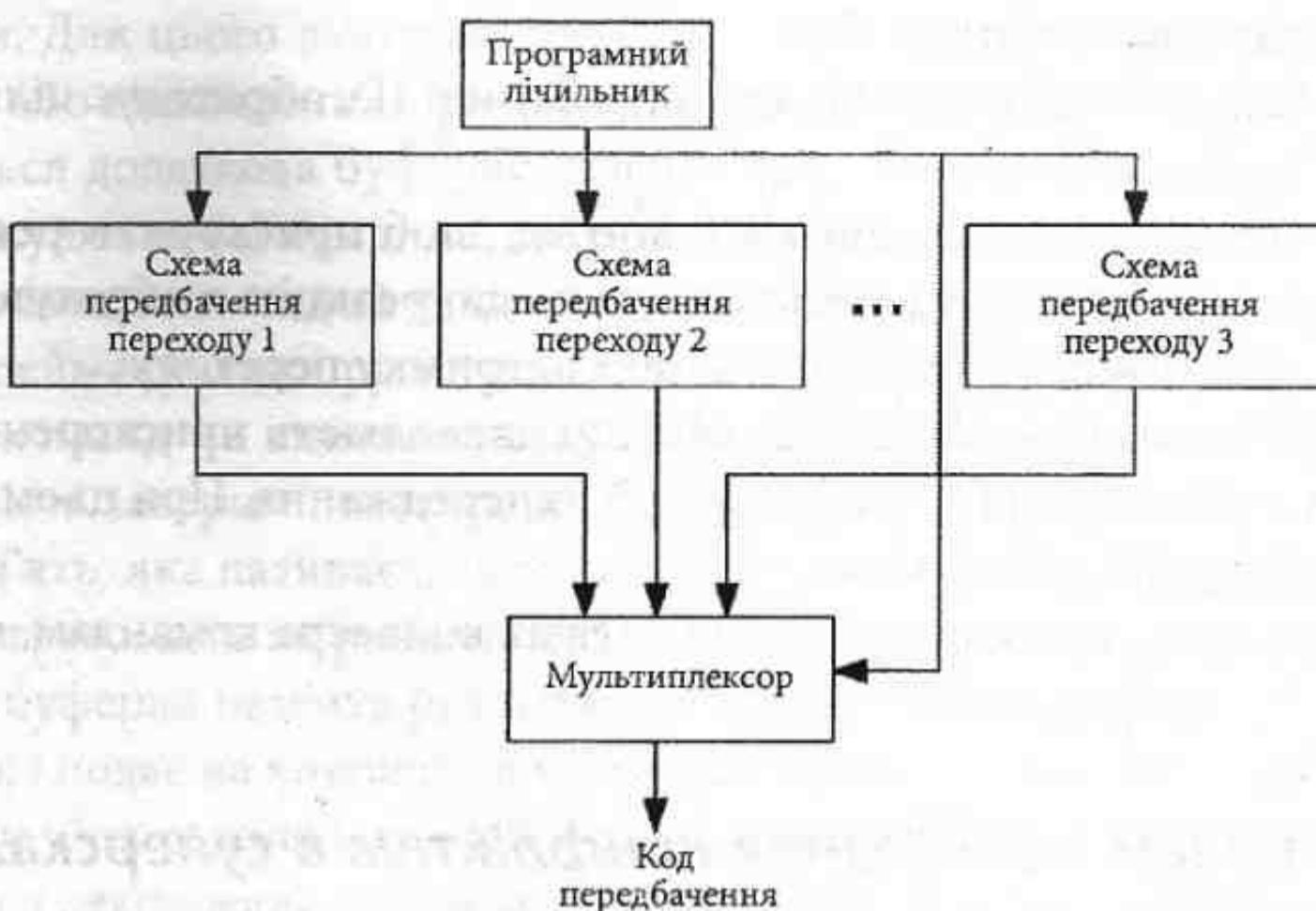


Рис. 5.21. Структура гібридної схеми передбачення переходу

Адресування конкретної схеми передбачення переходу та вибір схеми передбачення переходу здійснюються від програмного лічильника, тобто адресою команди, для якої здійснюється передбачення. Оновлення таблиць історії проводиться за раніше описанім правилом. Такі схеми передбачення є досить складними, але забезпечують найвищу точність передбачення.

5.4. Покращена структура комп’ютера із спрощеною системою команд

На основі проведеного вище аналізу конфліктів у конвеєрі та способів їх мінімізації можна провести покращання структури комп’ютера із спрощеною системою команд. В якості прикладу на рис. 5.22 подано покращену структуру тракту виконання команд комп’ютера DLX.

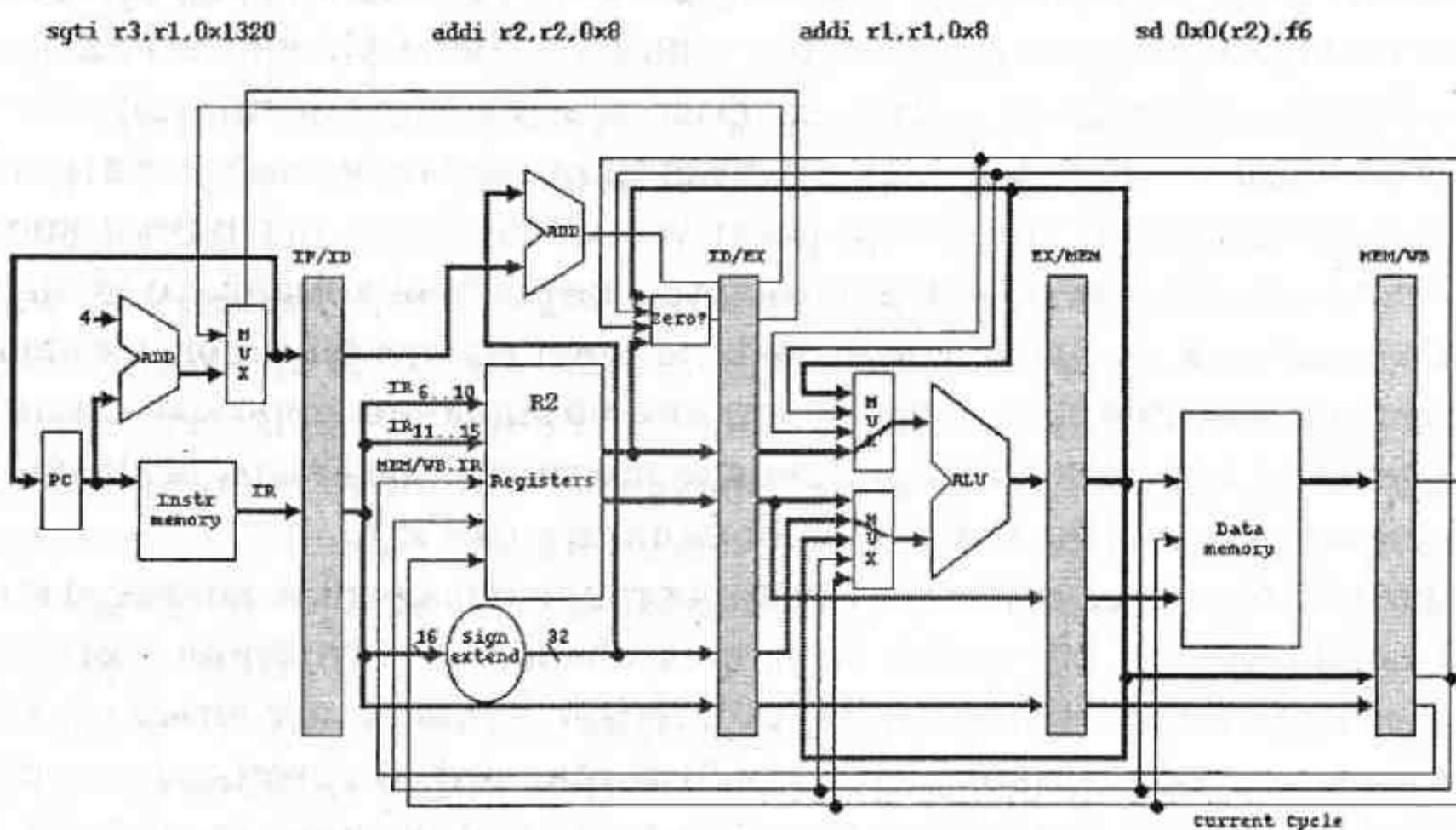


Рис. 5.22. Покращений конвеєр комп’ютера DLX

Структура додатково містить:

- суматор ADD, який прискорено, вже на сходинці ID, обраховує цільову адресу переходу;
- вузол Zero перенесено на один ярус вперед, щоб прискорити реакцію конвеєра на команду умовного переходу та синхронізувати цю реакцію з обрахуванням цільової адреси переходу. В результаті зменшено на такт затримку переходу;
- введено додаткові інформаційні шини, що дозволяють прискорено надсилати до вузлів процесора коди операндів при реалізації випередження. При цьому кількість входів мультиплексорів АЛП збільшено.

Кольорами відтінені відповідності станів вузлів конвеєра командам, які цей конвеєр опрацьовує.

5.5. Особливості запобігання конфліктам в суперскалярних процесорах

В суперскалярних процесорах, як ми вже бачили з їх розгляду в розділі 5, паралельно працює кілька конвеєрів команд. Тому в цих процесорах, як і в процесорах із простою системою команд, можливі раніше описані конфлікти при конвеєрному виконанні команд. Крім того, в них на додаток можливі конфлікти, які пов'язані з тим, що команди знаходяться в паралельних вітках. Тому тут ускладнюються питання запобігання раніше описаним структурним конфліктам, конфліктам за даними та конфліктам керування, і, крім того, добавляються конфлікти, пов'язані із забезпеченням впорядкованого поступлення команд на виконання та впорядкованого завершення команд.

Під впорядкованим поступленням команд та впорядкованим завершенням команд розуміють таким чином впорядковану черговість поступлення команд на виконання та черговість завершення команд, яка визначена програмою. В іншому випадку говорять про невпорядковане поступлення команд та невпорядковане завершення команд.

В перших суперскалярних процесорах типу Pentium було реалізоване впорядковане поступлення команд і впорядковане завершення команд. Такий підхід був досить простим, оскільки при виникненні конфліктів у одному з конвеєрів процесора призупинялась робота іншого конвеєра до зняття конфлікту, з тим, щоб не порушувати порядок поступлення та завершення команд. Однак це вимагало великих часових втрат.

Коли в суперскалярному процесорі реалізується стратегія підтримки впорядкованого поступлення команд та невпорядкованого завершення команд, то це дає можливість конвеєрам, в яких не відбулися конфлікти, продовжити опрацювання команд. Це дозволяє підвищити ефективність використання обладнання конвеєра команд. Однак при цьому потрібно забезпечити отримання коректних результатів, оскільки можливі некоректні записи результатів у регистровий файл та в пам'ять.

Іще більше підвищити ефективність використання обладнання конвеєра команд дозволяє реалізація стратегії підтримки невпорядкованого поступлення команд і невпорядкованого завершення команд. За цією стратегією команди подаються на виконання не в порядку їх поступлення в конвеєр, а при їх готовності до опрацювання, тобто коли наявні операнди та вільний функціональний вузол, в якому вони мають бути опрацювані. Для виконання цієї стратегії необхідно забезпечити коректність результату вико-

нання програми. Для цього використовується спосіб, який називається перевпорядкуванням команд, або відкладенням виконання команд. Для перевпорядкування команд в конвеєр вводиться додаткова буферна пам'ять, яка називається вікном команд. До цієї пам'яті завантажуються всі команди, які пройшли декодування, та, при необхідності запобігання конфліктам команд за даними при запису результатів до регістрового файла, виконується перейменування регистрів (див. п. 5.2.8). Вікно команд забезпечує відкладення передачі команд на виконання до готовності операндів і дозволяє забезпечити потрібну черговість завершення команд.

Буферна пам'ять, яка називається вікном команд, може бути реалізована двома способами – як інтегрована та як розподілена.

Інтегрована буферна пам'ять реалізується на основі асоціативної пам'яті. Вона оперативно виявляє і подає на виконання команди, для виконання яких є всі необхідні операнди та ресурси. Дляожної команди в цій пам'яті виділяється одна комірка. В цій комірці зберігається декодована команда, яка має наступні поля: декодоване поле операції (О), поля операндів (ПО), які вміщують або самі операнди, або адреси їх знаходження, поле, яке вказує місце розміщення результату (ПР), а також поле розрядів достовірності (РД). Для аналізу доступності та виконання розподілу команд до вільних функціональних пристрій (ФП) в процесорі використовується пристрій диспетчеризації, в якому є регистр занятості функціональних пристрій процесора (рис. 5.23).

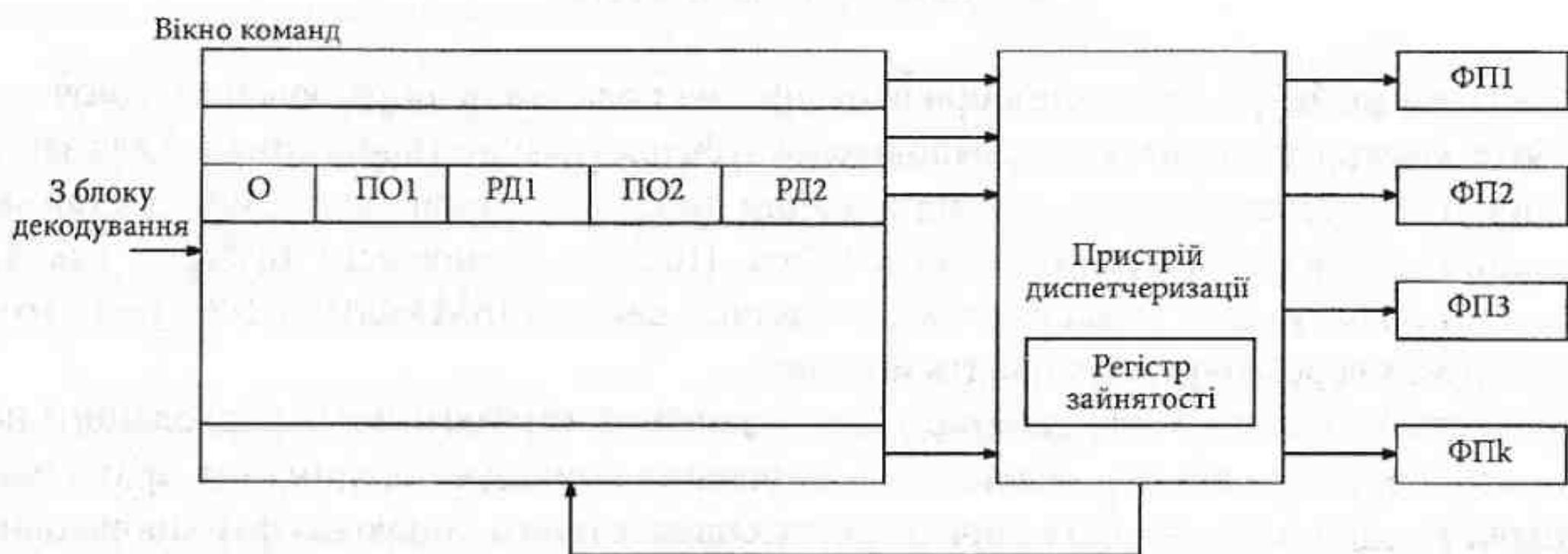


Рис. 5.23. Використання інтегрованої буферної пам'яті (вікна команд) для перевпорядкування команд

Команда зчитується з інтегрованої буферної пам'яті на виконання лише після того, коли в полях операндів ПО1 та ПО2 присутні значення операндів або їх адрес (тобто в полі розрядів достовірності РД1 та РД2 записані одиниці), та коли потрібні для її виконання функціональні пристрій є вільними (тобто в регистрі занятості записані одиниці). Результат виконання команди записується до відповідного регістру регістрового файла. Оновлення інформації про занятість функціональних пристрій процесора здійснюється в кожному його циклі.

Якщо вікно команд реалізується як розподілена буферна пам'ять, то на вході кожного функціонального блоку розміщується буфер декодованих команд, який називається блоком резервування (БР). Після вибірки та декодування команди поступають до того блоку резервування, в якому вони будуть виконуватися (рис. 5.24). Робота кожного

блоку резервування є аналогічною роботі інтегрованої буферної пам'яті, тобто команда поступає на виконання при готовності операндів та незайнятості функціонального пристрою.

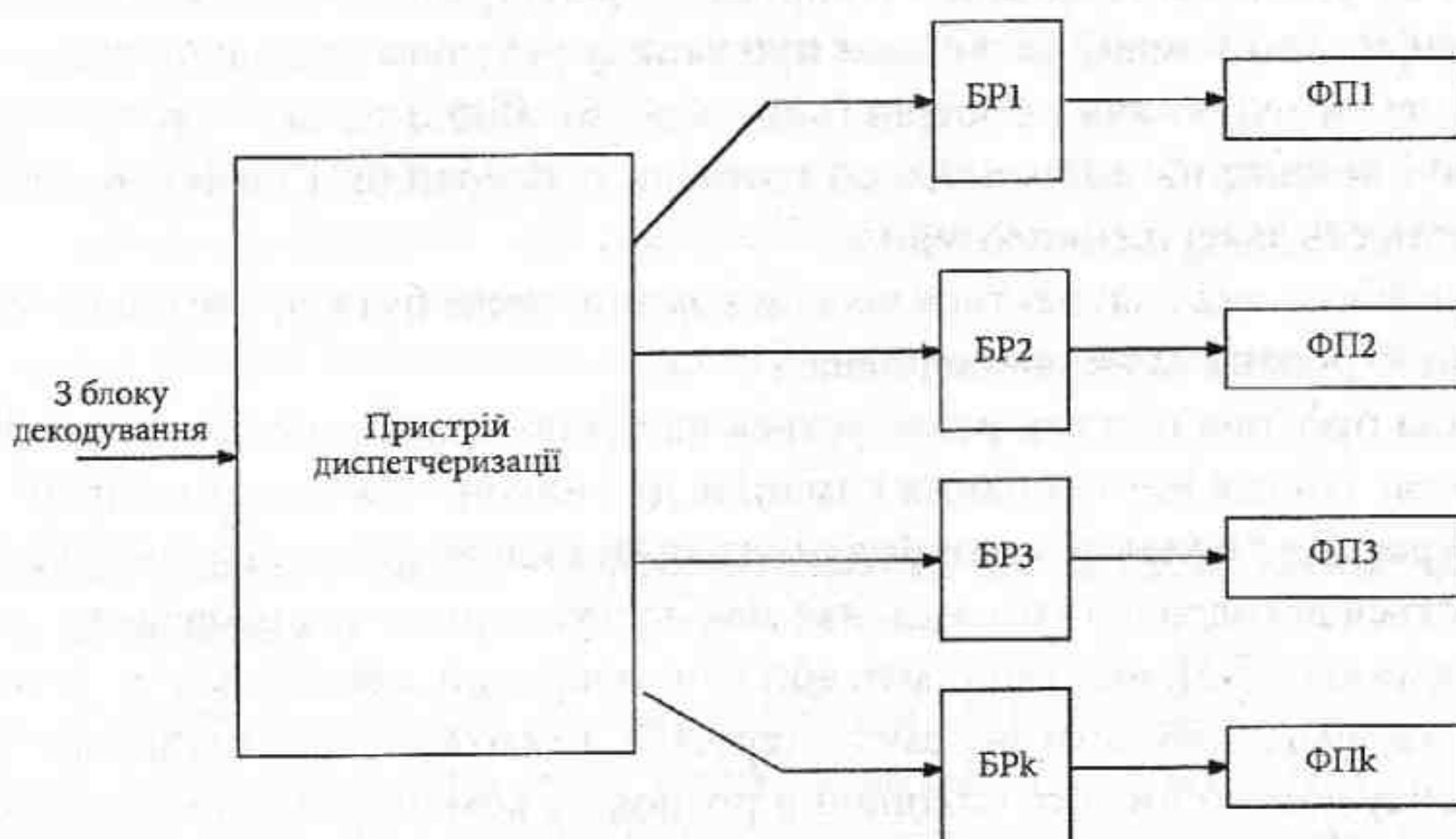


Рис. 5.24. Використання розподіленої буферної пам'яті (вікна команд) для переворядкування команд

Оскільки метод резервування орієнтований на подання на опрацювання одночасно декількох команд, використання розподіленої буферної пам'яті (вікна команд) для забезпечення переворядкуванням команд є значно простішим порівняно з використанням багатопортової інтегрованої буферної пам'яті. Подібну розподілену буферну пам'ять вперше запропонував Р. Томасуло в комп'ютерній системі IBM360/91 у 1967 році, тому описаний метод резервування носить його ім'я.

Важливим питанням для забезпечення виконання стратегії невпорядкованого поступлення команд та невпорядкованого завершення команд є підтримання правильної послідовності виконання команд при декількох паралельно працюючих функціональних пристроях. Вирішення цього питання було знайдене в 1988 році Смітом та Плескуном, які запропонували для цього використати буфер відновлення послідовності. До цього буфера команди записуються в порядку, який відповідає заданому програмою порядку їх зчитування, а зчитування команди з буфера дозволено тільки після завершення її виконання та коли всі попередні команди вже зчитані з буфера.

5.6. Комп'ютери з довгим форматом команд

Раніше розглянуті суперскалярні процесори характеризуються високою продуктивністю при забезпеченні безконфліктного виконання команд. Але це вимагає значних додаткових затрат обладнання. На рис. 5.25 показано кристал суперскалярного процесора MIPS R10000, на якому позначені вузли процесора та виділено апаратні засоби пристрою керування, які забезпечують невпорядковане виконання команд. Видно, що ці засоби зайняли більше 30 відсотків площини кристала.

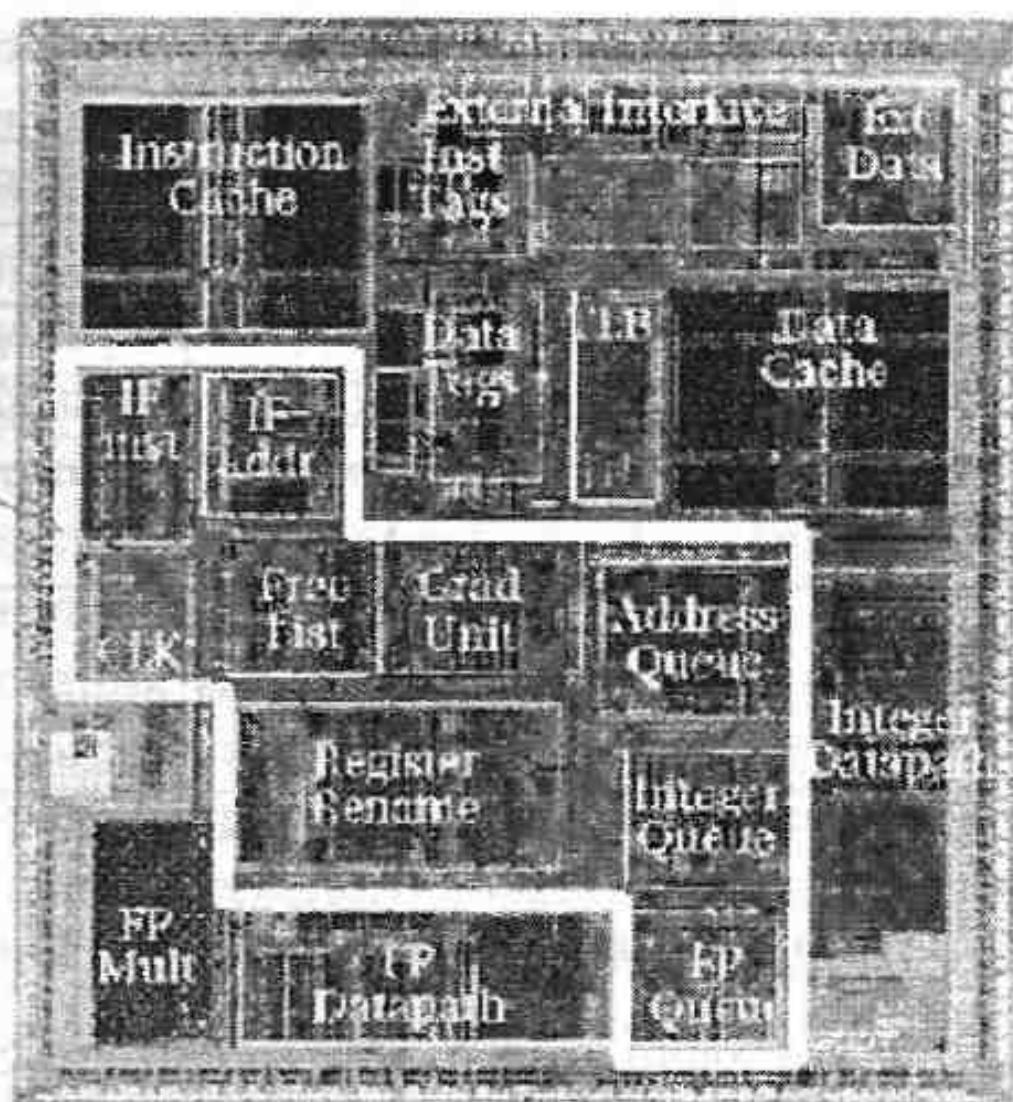


Рис. 5.25. Кристал суперскалярного процесора MIPS R10000, на якому позначені вузли процесора та виділено засоби керування невпорядкованим виконанням команд

Розглянемо далі архітектури комп’ютерів, у яких відсутні конфлікти команд. До них, зокрема, належать комп’ютери з довгим форматом команди

Архітектура комп’ютерів з довгим форматом команди (КДФК), англійський еквівалент VLIW (Very Long Instruction Word), бере свій початок від паралельного мікрокоду, що застосовувався в комп’ютерах CDC6600 і IBM 360/91. У 70-х роках багато комп’ютерних систем оснащувалися додатковими векторними процесорами обробки сигналів, що використовували довгий формат команди. Зокрема, до таких процесорів належали процесори AP-120B, AP-190L та інші фірми FPS.

Першими універсальними комп’ютерами з архітектурою КДФК стали міні-суперкомп’ютери, випущені на початку 1980-х років компаніями MultiFlow, Culler і Cydrome, але вони не мали комерційного успіху. Наприклад, комп’ютер компанії MultiFlow 7/300 використовував два арифметико-логічні пристрої для цілих чисел, два арифметико-логічні пристрої для чисел з рухомою комою і блок логічного галуження. Його 256-розрядний довгий формат команди містить поля для восьми 32-розрядних команд. В цих комп’ютерах були використані планувальник обчислень і програмна конвеєризація, які є основою технології компілятора КДФК.

Архітектура КДФК передбачає наявність багатьох незалежних функціональних пристрій. Для забезпечення паралельного виконання декілька команд пакуються в один пакет, який в подальшому будемо називати в’язанкою команд, та поступають на виконання. В’язанка команд може включати, наприклад, команди над числами з фіксованою та рухомою комою, команду звернення до пам’яті та команду переходу, як це показано на рис. 5.26. Така в’язанка команд матиме набір полів команд для кожного функціонального пристрію, що призводить до кратного збільшення її довжини порівняно з однією командою. Як видно з рисунка, кожне з цих полів керує відповідним функціональним пристроям, а обмін даними між пристроями здійснюється через інтегрований багатопортовий регистровий файл.



Рис. 5.26. Виконання в'язанки команд в незалежних функціональних пристроях

Завдання ефективного планування паралельного виконання команд в комп'ютері з довгим форматом команди повністю лягає на компілятор. Компілятор з послідовності команд початкової програми генерує в'язанку команд шляхом проглядання програми як в межах, так і за межами лінійних ділянок програми без галужень. Для забезпечення коректності виконання операцій компілятор має бути здатним визначати можливість появи конфліктів при виконанні команд. У певних ситуаціях, коли компілятор не може зробити заключення, що конфлікт не відбудеться, наприклад, у випадку звернення до масиву, коли індекс обчислюється під час виконання програми, операції не можуть плануватися для паралельного виконання. Відповідно це призводить до зниження продуктивності.

Компілятор визначає ділянку програми без циклів, яка стає кандидатом для формування в'язанки команд. Для збільшення розміру тіла циклу широко використовується методика розкручування циклів, що призводить до утворення великих фрагментів програми, що не містять зворотних дуг.

Якщо виконанню підлягає програма, що має тільки переходи вперед, компілятор робить евристичний прогноз вибору умовних гілок. Шлях, що має найбільшу вірогідність виконання (його називають трасою), використовується для оптимізації, що проводиться з врахуванням конфліктів за даними між командами і обмежень за апаратними ресурсами. Під час планування генерується в'язанка команд. Всі команди, які входять до в'язанки команд, видаються одночасно і виконуються паралельно. Після обробки першої траси планується наступний шлях, що має найбільшу вірогідність виконання (попередня траса більше не розглядається). Процес пакування команд послідовної програми у в'язанки команд продовжується до тих пір, поки не буде оптимізована вся програма.

Ключовою умовою досягнення ефективної роботи комп'ютера з довгим форматом команди є коректний прогноз вибору умовних гілок. Відмічено, наприклад, що прогноз умовних гілок для наукових програм часто виявляється точним. Повернення назад є у всіх ітераціях циклу, за винятком останньої. Таким чином, прогноз буде коректний в більшості випадків. Інші умовні гілки, наприклад, гілка обробки переповнення і перевірки граничних умов (вихід за межі масиву), також є надійно передбачуваними.

З погляду архітектурних ідей комп'ютер з довгим форматом команди можна розглядати як розширення архітектури комп'ютера з простою системою команд, скільки тут

також довжина команди є сталою, апаратні ресурси плануються статично, та віддається перевага програмному вирішенню конфліктів.

В архітектурі комп'ютера з довгим форматом команди, крім того, принцип заміни керування під час виконання програми на планування під час компіляції поширений і на систему пам'яті. Для забезпечення зайнятості конвеєрних функціональних пристрій тут необхідна пам'ять з високою пропускною спроможністю. Одним із сучасних підходів до збільшення пропускної спроможності пам'яті є використання розшарування пам'яті, причому можливі конфлікти доступу до пам'яті визначає спеціальний модуль компілятора – модуль запобігання конфліктам.

Таким чином, розпаралелювання у комп'ютерах з довгим форматом команди виконується виключно на етапі компіляції під час формування в язанок команд, тобто статично. Так як неможливо наперед передбачити розв'язання усіх видів залежностей, відкомпільовані програми вимагають ретельного налагоджування. Негативно впливають також ефекти невпорядкованого завершення виконання команд. Саме тому надійність виконання програми тут зменшена порівняно з суперскалярним варіантом архітектури комп'ютера. Тобто, цей підхід дозволяє досягти максимуму продуктивності, але є прийнятним лише у певних застосуваннях комп'ютерних засобів. Перевага КДФК в потенційній продуктивності найбільш відчутна в серверних задачах, де паралельно опрацьовують декілька процесів (ниток), в наукових задачах, задачах тривимірної візуалізації та в задачах обробки сигналів (де, зокрема, застосовуються процесори ADSP21XX фірми Analog Devices та TMS320C6X фірми Texas Instruments, які належать до вказаної архітектури).

Архітектуру КДФК запроваджено у новітніх процесорах Alpha фірми DEC та IA-64 фірми Intel. Останній процесор оптимізовано для виконання серверних задач. Як приклад на рис. 5.27 приведено структуру ядра процесора TMS320C6X фірми Texas Instruments, в якому використано довгий формат команди. В'язанка команд цього процесора складається з восьми 32-розрядних команд. Тут використано два тракти обробки даних A та B, кожний з яких має по 4 функціональні блоки та свій багатопортовий регистровий файл.

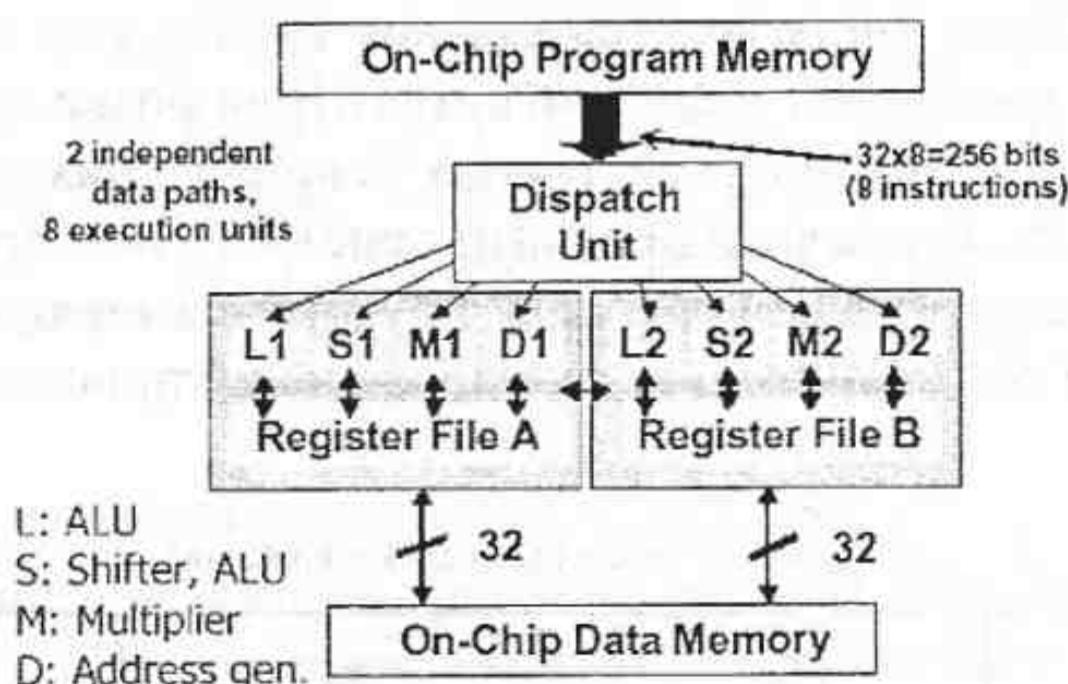


Рис. 5.27. Структура ядра процесора TMS320C6X фірми Texas Instruments

Тракти обробки даних A і B є однотипними та мають в своєму складі наступні функціональні пристрій: L – АЛП, S – пристрій зсуву та АЛП, M – перемножувач, D – формувач адрес пам'яті. В'язанка команд зчитується з пам'яті команд (On-Chip Program Memory) та поступає до блоку диспетчеризації (Dispatch Unit), який здійснює розподіл команд у відповідні функціональні пристрій.

5.7. Комп'ютери з комбінованою архітектурою

При реалізації архітектури КДФК виникають серйозні проблеми. Компілятор цього комп'ютера повинен в деталях враховувати внутрішні особливості процесора, аж до організації роботи його функціональних пристрій. Як наслідок, при випуску нової версії комп'ютера з більшою кількістю функціональних пристрій доводиться радикально переписувати і компілятор. Інша проблема – це за своєю суттю статична природа оптимізації, яку забезпечує компілятор КДФК. Важко передбачити, як, наприклад, поведеться компілятор, коли зіткнеться під час компіляції з непередбаченими динамічними ситуаціями, такими як очікування введення/виведення. Тому розробники сучасних швидко-діючих комп'ютерів починають відходити від чистої архітектури КДФК.

У 2000 році корпорація Трансмета випустила процесор Crusoe, сумісний із системою команд процесора x86, який характеризується високою продуктивністю та зниженою споживаною потужністю. Покращання характеристик досягнуто шляхом заміни відчутної частки апаратури процесора допоміжною, але обов'язковою, програмною оболонкою. Процесор Crusoe складається з апаратного ядра та програмної оболонки, яка формує програмний код для цього ядра. Ядром є нескладний КДФК, здатний виконувати до чотирьох простих команд в кожному такті. Доповнення апаратного ядра програмною оболонкою для формування програмного коду створює ефект присутності повноцінного набору апаратних засобів архітектури x86. Програмну оболонку формування коду називають *Code Morphing™* тому, що вона динамічно відбиває (“morphs”) зовнішні (складні) команди процесора x86 у внутрішні (спрощені) в'язанки команд апаратного ядра.

Технологія *Code Morphing™* фірми Трансмета змінює традиційні методи проектування процесорів. Практична апробація того, що серійні процесори можна реалізувати як гібрид програмних і апаратних засобів, суттєво розширила можливості проектувальників по вибору та впровадженню оптимальних рішень. По перше, паралельна робота проектувальників апаратного ядра процесора та системних програмістів, які розробляють програмну оболонку, сприяє скороченню термінів розробки. По друге, зовнішня програмна оболонка формування коду із вязанок команд для ядра процесора забезпечує незалежний від прихованої апаратної архітектури розвиток програмних засобів комп'ютерної системи в цілому тобто системних та прикладних програм.

Ядро процесора включає два функціональні пристрій для операцій з фіксованою комою, функціональний пристрій для операцій з рухомою комою, функціональний пристрій звернення до пам'яті (load/store) та функціональний пристрій виконання умовних переходів (рис. 5.28).



Рис. 5.28. Загальна структура ядра процесора Crusoe

Тут ADD – операція додавання з фіксованою комою, FADD – операцією додавання з рухомою комою, LD – завантаження подвійного слова, а BRCC – умовний перехід за деякою умовою (Conditional Code Branch).

В'язанка команд процесора Crusoe отримала назву молекули. Вона має довжину 64/128 бітів та вміщує чотири команди (четири атоми). Усі атоми в межах молекули виконуються паралельно. Формат молекули прямо визначає зв'язок атома із конкретним функціональним пристроєм, що значно і спрощує, і прискорює апаратні засоби. Самі молекули процесор опрацьовує із збереженням черговості, тобто без використання складних апаратних засобів невпорядкованого опрацювання. Для забезпечення ефективного використання процесора мінімізується кількість незаповнених атомів, а у випадках несумісності атомів за паралельністю, на місце відсутнього атома вставляється пуста операція.

Програма Code Morphing є фундаментальною системою динамічної трансляції, що транслює команди однієї архітектури (в даному випадку КССК процесора x86) в команди іншої архітектури (КДФК). Програму Code Morphing розміщено в ПЗП процесора і вона є першою програмою, що виконується при завантаженні процесора. Рис. 5.29 ілюструє зв'язок між кодом процесора x86, програмою Code Morphing та ядром процесора Crusoe.

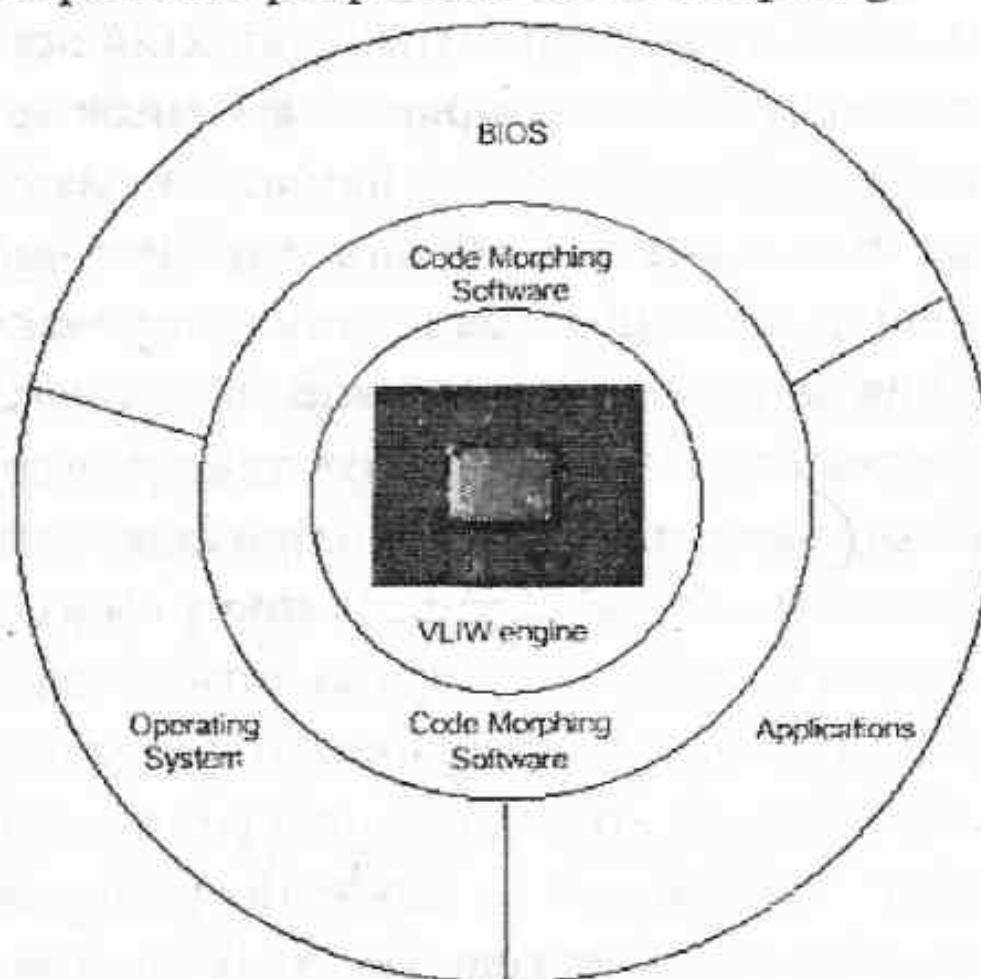


Рис. 5.29. КДФК (VLIW engine), Code Morphing, BIOS, операційна система та прикладні програми в кодах процесора x86

Завдяки тому, що програма формування коду КДФК, а саме, Code Morphing, ізоляє програми процесора x86 (системні та прикладні разом із BIOS і операційною системою) від апаратури процесора Crusoe із притаманною йому системою команд КДФК, цю систему команд можна змінювати без жодного впливу на систему команд процесора x86. Але при зміні архітектури КДФК певних змін зазнає і програма Code Morphing. Проте зміни до програми формування коду Code Morphing треба виконувати лише один раз на кожну зміну архітектури ядра.

Приховання реальної архітектури рівня машинних команд за шаром фіrmової програми Code Morphing дозволяє уникнути проблем, притаманних чистим КДФК. Чисті КДФК змушують розробника компілятора враховувати усі деталі роботи конвеєра. При

цьому найменші зміни у конвеєрі вимагають перепроектування компілятора (не звичайний, а оптимізуючий), що є надто складною ресурсомісткою та кропіткою працею. Зазначеної проблеми для процесора Crusoe просто не існує. З погляду системних програм Crusoe виглядає як стандартний процесор x86.

Описана технологія віртуалізації архітектури комп'ютера зі складною системою команд є принципово новим і ефективним методом декомпозиції та вирішення складних завдань проектування сучасних комп'ютерів. Запропонований фірмою Трансмета програмно-апаратний метод проектування забезпечив вдалий розподіл функцій процесора між його програмною та апаратною частинами, і, тим самим, дозволив знайти компроміс між вартістю та складністю, між продуктивністю та споживаною потужністю.

Програма Code Morphing формування коду КДФК транслює групи команд процесора x86, а не кожну відокремлену команду, як у суперскалярному процесорі. Зрозуміло, що опрацювання групи команд розширяє діапазон дій програмного транслятора, що дає йому можливість враховувати семантику фрагмента коду, разом із притаманною фрагментові кореляцією поміж сусідніми командами. Саме це обумовлює перевагу трансляційної технології Crusoe. Більше того, суперскалярний процесор транслює команду кожного разу, як її виконує. Crusoe виконує трансляцію команди одноразово, зберігаючи результат трансляції у так званій трансляційній кеш пам'яті. Коли виконання фрагмента програми процесора x86 повторюється, Crusoe не виконує повторну трансляцію, а забирає потрібне з трансляційної кеш пам'яті. Зрозуміло, що "компільований" фрагмент коду процесора x86 виконується швидше від "інтерпретованого".

Програмна, а не апаратна, реалізація трансляції відкриває нові можливості. Адже апаратна реалізація складних алгоритмів трансляції збільшує кількість транзисторів на кристалі та споживану потужність. Crusoe виконує лише одноразову повільну трансляцію при першому проході програми, а всі наступні проходи здійснюють швидко, що дозволяє значно підвищити складність та ефективність програмно реалізованих алгоритмів трансляції, підвищивши тим самим характеристики комп'ютерної системи.

Зрозуміло, що програма Code Morphing використовує властивість локалізації (locality of reference) вибірань програмних кодів будь-якої універсальної машини. Апаратура процесора Crusoe опрацьовує отриманий від Code Morphing молекулярний код не хаотично, а впорядковано, за природною чергою. Це гарантує впорядковане опрацювання оригінальних команд процесора x86. Саме молекули однозначно визначають паралелізм рівня машинних команд, тому апаратна реалізація процедури паралельного опрацювання спрощується.

5.8. Комп'ютери з явним паралелізмом виконання команд

Як ми вже побачили з матеріалу даного розділу, планування порядку обчислень є досить важким завданням, яке доводиться вирішувати при проектуванні сучасного комп'ютера. У суперскалярній архітектурі для розпізнавання залежностей між командами застосовуються досить складні апаратні рішення (наприклад, в P6 і наступних процесорах фірми Intel для цього використовується буфер переворядковування команд – Reorder Buffer). Проте розміри такого апаратного планувальника при збільшенні кількості функціональних пристрій зростають в геометричній прогресії. Тому суперскалярні

проекти зупинилися на відмітці 5–6 оброблюваних за цикл команд. Навіть КДФК теж далеко не завжди можуть забезпечити повне заповнення в'язанок команд – реальне завантаження близько 6–7 команд в такті.

Вище зазначені проблеми призначена вирішувати нова архітектура із назвою EPIC (Explicitly Parallel Instruction Set Computing).

Класичний процесор, переважно, не спроможний визначити взаємовплив (залежності даних, керування і структури) поміж віддаленими командами первинного (поки що не розпаралеленого) потоку команд. Адже процесор аналізує лише той фрагмент потоку команд, що розташований безпосередньо у процесорному апаратному, тому і невеликому, буфері команд. З іншого боку, програма-компілятор виконуваного коду має значно ширше поле зору та практично необмежений час (ще до виконання програми процесором), аби наперед проаналізувати первинний програмний код на предмет виявлення вказаних вище залежностей та оптимізувати цей код під потрібну архітектуру. В цілому, тут ми маємо ситуацію, коли бажано все, що можна, підготувати заздалегідь (зрозуміло, статично, під час компіляції), а не приймати складних апаратних динамічних рішень в реальному часі (при виконанні процесором вже остаточно скомпільованої програми, із притаманними швидкими рішеннями, помилками та з відповідними часовими витратами на їх виправлення).

Архітектура EPIC передбачає пряме розпаралелювання виконання програми, тобто компілятор мусить повідомляти процесор про те, яка частина коду може виконуватися паралельно. Оптимізований за попереднім означенням компілятор EPIC аналізує програмний код, аби визначати, де та коли відбудуться/не відбудуться умовні переходи та знайти і позначити ті частини програмного коду, які можна виконувати паралельно.

Прикладом процесора з архітектурою EPIC є процесор IA-64 фірми Intel. В цьому процесорі команди, як і в архітектурі КДСК, пакуються компілятором в 128-роздрядні в'язанки команд. Кожна в'язанка команд процесора IA-64 містить три команди та шаблон, як це показано на рис. 5.30.

127				0
Команда 2	Команда 1	Команда 0	Шаблон	

Рис. 5.30. В'язанка команд процесора IA-64

В шаблоні вказується залежність між командами в одній в'язанці та між в'язанками, тобто вона вказує, чи можна одночасно виконувати i -ту ($i = 0, 1, 2$) команду в'язанки та одночасно з j -ю ($j = 0, 1, 2$) командою в'язанки п. Кожній в'язанці в процесорі виділяється три функціональних пристрої, тобто кожній команді виділяється один пристрій. Вміст поля шаблону встановлюється або при генеруванні коду компілятором, або безпосередньо системним програмістом, що пише мовою асемблер. Процес генерації коду виконують так, аби гарантовано позбутися конфліктів типу RAW, WAW в межах командної групи.

Кожна з трьох команд в'язанки має формат, приведений на рис. 5.31.

39	27 26	21 20	14 13	7 6	0
Код операції	Предикат	Рг операнда 1	Рг операнда 2	Рг результату	

Рис. 5.31. Форматожної з трьох команд в'язанки

До складу команди входять наступні поля: коду операції, предиката, номери регістрів двох операндів та регістра результату. Розрядності кожного поля вказані на рисунку.

Поле предиката вказує номер регістра предиката, яких в процесорі є 64. Предикація – це спосіб обробки умовних переходів. В процесорі IA-64 виконуються обидві вітки переходу. Суть способу предикації полягає в наступному. Командам різних гілок одного умовного переходу виділяються різні регістри предиката. Ці команди виконуються на функціональних пристроях процесора, а їх результати записуються до пам'яті тільки після визначення вмісту регістрів предиката, тобто після обчислення умови переходу. Вмісту регістрів вітки переходу, якій відповідає умова переходу, присвоюється значення 1, а вмісту регістрів вітки переходу, якій не відповідає умова переходу, присвоюється значення 0. Процесор перевіряє вміст регістрів предикату і записує в пам'ять результати тільки тих команд, які вказують на регістри предикатів з одиничним вмістом.

Архітектура IA-64 підвищує рівень паралельного виконання команд за рахунок того, що вона дозволяє на рівні мови асемблеру прямо вказувати на паралелізм, реалізує в'язанки, кожна з яких містить три виконувані команди та містить множину надлишкових програмно-недосяжних регістрів, що дублюють операнди поточних команд, запобігаючи тим самим завчасному перезапису цих операндів.

5.9. Короткий зміст розділу

Розглянуті конфлікти в конвеєрі команд та методи їх усунення, оскільки вони знижують продуктивність конвеєра, яка могла б бути досягнута в ньому в ідеальному випадку. Більше того, конфлікти можуть звести нанівець всі затрати на створення конвеєра команд. Проведено аналіз методів запобігання трьом класам конфліктів: структурних, які виникають з причини браку ресурсів, коли апаратні засоби не можуть підтримувати всі можливі комбінації команд в режимі одночасного виконання з перекриттям, конфліктів за даними, що виникають у разі, коли виконання наступної команди залежить від результату виконання попередньої команди, та конфліктів керування, які виникають при конвеєризації команд передачі керування, які змінюють значення лічильника команд. Наведено приклади структур конвеєрних процесорів, у яких зменшено ймовірність виникнення конфліктів.

Розглянуто особливості запобігання конфліктам в суперскалярних процесорах, які є наступним кроком в побудові високопродуктивних процесорів. Суперскалярний процесор має кілька функціональних блоків і виконує кілька команд за один такт, тобто в такому процесорі одна команда виконується менше як за один такт. Прикладами суперскалярних процесорів є PowerPC фірми IBM, UltraSparc фірми Sun, Alpha фірми DEC. Але методи запобігання конфліктам в таких процесорах є ще складнішими, ніж у конвеєрних процесорах, що вимагає відповідного ускладнення апаратних засобів.

Були розглянуті архітектури комп'ютерів, в яких відсутні конфлікти команд, а саме: комп'ютери з довгим форматом команди, а також комбіновані архітектури, в яких поєднано архітектури КПСК та КДФК.

Проблеми забезпечення динамічного планування виконання команд привели розробників до архітектури комп'ютера з явним паралелізмом EPIC, прикладом якої стала розробка IA-64 фірми Intel. Комп'ютери цієї архітектури опрацьовують паралельно в'язанку команд, яка вказує декілька операцій, що можуть виконуватися паралельно.

Система команд цієї архітектури тісно пов'язана з будовою компілятора, оскільки планування паралельного виконання команд тут покладено на компілятор, який здійснює цю роботу перед виконанням програми в комп'ютері.

Для зменшення впливу умовних переходів на продуктивність конвеєра в процесорі IA-64 введено предикатні команди. В цьому процесорі всі команди виконуються, але результати їх виконання записуються до реєстрового файла лише тоді, коли розряд предиката рівний 1. Результатом є те, що не потрібно зупиняти конвеєр до вияснення умови переходу, хоча виконується більша кількість команд.

5.10. Література для подальшого читання

Конфлікти в конвеєрі команд та методи їх усунення розглянуті в роботах [7, 8, 13, 14, 16, 18–21]. В роботах [3, 4, 30–33] проведено аналіз методів запобігання трьох класів конфліктів: структурних, конфліктів за даними та конфліктів керування. Опис симулятора WinDLX є на web-сторінці www.

В роботах [9, 10, 22–26] розглянуто особливості запобігання конфліктам в суперскалярних процесорах. Для аналізу особливостей реалізації засобів запобігання конфліктам в суперскалярних процесорах PowerPC фірми IBM, UltraSparc фірми Sun, Alpha фірми DEC та інших доцільно пошукати їх описи на web-сторінках цих фірм. Використання розподіленої буферної пам'яті (вікна команд) для перевпорядкування команд запропоновано в роботах [1, 27].

Обмеження паралелізму рівня команд проаналізовано в [28, 29]. В роботах [3, 5, 6, 12, 15] детально розглянуті архітектури комп'ютерів, у яких відсутні конфлікти команд, а саме комп'ютерів з довгим форматом команди. Зокрема в роботі [2] розглянуті питання побудови перших процесорів обробки сигналів AP-120B фірми FPS з архітектурою КДФК. Принципи побудови компіляторів КДФК можна знайти в [4, 11, 17].

Інформацію про комбіновані архітектури, в яких поєднано архітектури КПСК та КДФК, можна знайти в [5].

Архітектура комп'ютера з явним паралелізмом EPIC описана в [7].

5.11. Література до розділу 5

1. Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo [1967]. “The IBM 360 Model 91: Processor philosophy and instruction handling”, IBM J. Research and Development 11:1 (January), 8–24.
2. Charlesworth, A. E. [1981]. “An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family”, Computer 14:12 (December), 12–30.
3. COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND P. K. RODMAN [1987]. “A VLIW architecture for a trace scheduling compiler”, Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM (March), Palo Alto, Calif., 180–192.
4. Ellis, J. R. [1986]. Bulldog: A Compiler for VLIW Architectures, MIT Press, Cambridge, Mass.
5. FISHER, J. A. [1981]. “Trace scheduling: A technique for global microcode compaction”, IEEE Trans, on Computers 30:7 (July), 478^t90.
6. FISHER, J. A. [1983]. “Very long instruction word architectures and ELI-512”, Proc. Tenth Symposium on Computer Architecture (June), Stockholm, 140–150.
7. Fisher, J. A. and S. M. Freudenberger [1992]. “Predicting conditional branches from previous runs of a program”, Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM (October), Boston, 85–95.

8. Hwu, W.-M. and Y. Patt [1986]. "HPSm, a high performance restricted data flow architecture having minimum functionality", Proc. 13th Symposium on Computer Architecture (June), Tokyo, 297–307.
9. Johnson, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.
10. JOUPPI, N. P. AND D. W. WALL [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors", Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM (April), Boston, 272–282.
11. Lam, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW processors", SIGPLAN Conf. on Programming Language Design and Implementation, ACM (June), Atlanta, Ga., 318–328.
12. Mahlke, S. A., W. Y. Chen, W.-M. Hwu, B. R. Rau, and M. S. Schlansker [1992]. "Sentinel scheduling for VLIE and superscalar processors", Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems (October), Boston, IEEE/ACM, 238–247.
13. McFarling, S. [1993] "Combining branch predictors", WRL Technical Note TN-36 (June), Digital Western Research Laboratory, Palo Alto, Calif.
14. McFarling, S. and J. Hennessy [1986]. "Reducing the cost of branches", Proc. 13th Symposium on Computer Architecture (June), Tokyo, 396–403.
15. NICOLAU, A. AND J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures", IEEE Trans, on Computers C-33:11 (November), 968–976.
16. Pan, S.-T., K. So, and J. T. Rameh [1992]. "Improving the accuracy of dynamic branch prediction using branch correlation", Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM (October), Boston, 76–84.
17. RAU, B. R., C D. GLAESER, AND R. L. PICARD [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support", Proc. Ninth Symposium on Computer Architecture (April), 131–139.
18. Riseman, E. M. and C. C Foster [1972]. "Percolation of code to enhance parallel dispatching and execution", IEEE Trans, on Computers C-21:12 (December), 1411–1415.
19. SMITH, A. and J. LEE [1984]. "Branch prediction strategies and branch-target buffer design", Computer 17:1 (January), 6–22.
20. Smith, J. E. [1981]. "A study of branch prediction strategies", Proc. Eighth Symposium on Computer Architecture (May), Minneapolis, 135–148.
21. Smith, J. E. and A. R. Pleszkun [1988]. "Implementing precise interrupts in pipelined processors", IEEE Trans, on Computers 37:5 (May), 562–573. This paper is based on an earlier paper that appeared in Proc. 12th Symposium on Computer Architecture, June 1988.
22. Smith, M. D., M. Horowitz, and M. S. Lam [1992]. "Efficient superscalar performance through boosting", Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems (October), Boston, IEEE/ACM, 248–259.
23. Smith, M. D., M. Johnson, and M. A. Horowitz [1989]. "Limits on multiple instruction issue".
24. SOHI, G. S. [1990]. "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers", IEEE Trans, on Computers 39:3 (March), 349–359.
25. SOHI, G. S. AND S. Vajapeyam [1989]. "Tradeoffs in instruction format design for horizontal architectures", Proc. Third Conf. on Architectural Support for Programming languages and Operating Systems, IEEE/ACM (April), Boston, 15–25.
26. THORLIN, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers", Proc. Spring Joint Computer Conf. 27.
27. TOMASULO, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units", IBM J. Research and Development 11:1 (January), 25–33.
28. WALL, D. W. [1991]. "Limits of instruction-level parallelism", Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems (April), Santa Clara, Calif., IEEE/ ACM, 248–259.

29. Wall, D. W. [1993]. Limits of Instruction-Level Parallelism, Research Rep. 93/6, Western Research Laboratory, Digital Equipment Corp. (November).
30. WEISS, S. and J. E. Smith [1984]. "Instruction issue logic for pipelined supercomputers", Proc. 11th Symposium on Computer Architecture (June), Ann Arbor, Mich., 110–118.
31. WEISS, S. and J. E. SMITH [1987]. "A study of scalar compilation techniques for pipelined supercomputers", Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems (March), IEEE/ACM, Palo Alto, Calif., 105–109.
32. Yeh, T. and Y. N. Patt [1992]. "Alternative implementations of two-level adaptive branch prediction", Proc. 19th Symposium on Computer Architecture (May), Gold Coast, Australia, 124–134.
33. YEH, T. AND Y. N. Patt [1993]. "A comparison of dynamic branch predictors that use two levels of branch history", Proc. 20th Symposium on Computer Architecture (May), San Diego, 257–266.

5.12. Питання до розділу 5

1. Назвіть причини необхідності забезпечення ефективного виконання команд в процесорі.
2. Назвіть три класи конфліктів у конвеєрі команд та причини їх появи.
3. Які є дві групи структурних конфліктів?
4. Наведіть приклад структурних конфліктів, які виникають через потребу порушення тактової частоти роботи конвеєра.
5. Наведіть приклад структурних конфліктів, які виникають у зв'язку з необхідністю очікування на звільнення ресурсів комп'ютера.
6. Чому розробники допускають наявність структурних конфліктів?
7. Яка причина створення процесорів з неконвеєрними функціональними пристроями?
8. На який час потрібно призупинити роботу конвеєра команд при появі структурних конфліктів?
9. Які є способи вирішення структурних конфліктів?
10. Коли виникає конфлікт за даними?
11. Назвіть три можливі конфлікти за даними.
12. Поясніть суть конфлікту "читання після запису".
13. Поясніть суть конфлікту "запис після читання".
14. Поясніть суть конфлікту "запис після запису".
15. Які можливі конфлікти за даними?
16. Які є методи зменшення впливу залежностей між даними на роботу конвеєра команд?
17. Що дає призупинення роботи конвеєра при виявленні конфлікту за даними?
18. Що дає застосування випереджуального пересилання при виявленні конфлікту за даними?
19. Як реалізується в конвеєрі команд випереджуальне пересилання?
20. Чи завжди є можливим випереджуальне пересилання?
21. Приведіть приклади можливих та неможливих випереджуальних пересилань.
22. Що роблять, оптимізуючи компілятори, щоб не допустити конфліктів за даними?
23. Які є ознаки наявності конфліктів за даними?
24. Для яких частин програми є ефективною статична диспетчеризація послідовності команд під час компіляції?
25. Як здійснюється динамічна диспетчеризація послідовності команд у програмі під час компіляції?
26. Поясніть суть методу перейменування реєстрів.
27. Які є типи конфліктів керування?
28. Назвіть способи зниження втрат на вибірку команд переходу.
29. Поясніть суть способу обчислення виконавчої адреси команди переходу в ярусі декодування команди.

30. Поясніть суть способу використання буфера адрес переходів.
31. Поясніть суть способу використання буфера команд переходів.
32. Поясніть суть способу використання буфера циклу.
33. Назвіть способи зниження втрат на виконання команд умовного переходу.
34. Поясніть суть способу введення буфера попередньої вибірки з метою зниження втрат на виконання команд умовного переходу.
35. Поясніть суть способу дублювання початкових ярусів конвеєра з метою зниження втрат на виконання команд умовного переходу.
36. Поясніть суть способу затримки переходу з метою зниження втрат на виконання команд умовного переходу.
37. Поясніть суть способу статичного передбачення переходу з метою зниження втрат на виконання команд умовного переходу.
38. Назвіть методи статичного передбачення умовного переходу.
39. Поясніть суть методу повернення, який застосовується при статичному передбаченні умовного переходу.
40. Поясніть суть методу профілювання, який застосовується при статичному передбаченні умовного переходу.
41. Поясніть суть методу статичного передбачення умовного переходу, за яким результат переходу визначається кодом операції команди переходу.
42. Поясніть суть методу статичного передбачення умовного переходу, за яким результат переходу визначається напрямом переходу.
43. Поясніть суть динамічного передбачення переходу.
44. Що таке таблиця історії переходів? Як вона реалізується?
45. Наведіть однорівневу схему передбачення переходу з формуванням адреси таблиці історії переходів в програмному лічильнику.
46. Наведіть однорівневу схему передбачення переходу з формуванням адреси таблиці історії переходів у регистрі глобальної історії.
47. Наведіть однорівневу схему передбачення переходу з комбінованим формуванням адреси таблиці історії переходів в програмному лічильнику та в регистрі глобальної історії.
48. Наведіть дворівневу схему передбачення переходу з використанням таблиці локальної історії.
49. Наведіть структуру гібридної схеми передбачення переходу.
50. Проаналізуйте тотожність та розбіжність КДФК і суперскалярної архітектур.
51. Визначте місце суперскалярних і КДФК архітектур в ієрархії сучасних комп'ютерів.
52. Визначте та поясніть основні чинники, що обмежують ефективність КДФК архітектури.
53. Наведіть основні ідеї, покладені в основу архітектури EPIC.