Министерство образования и науки Российской Федерации

Федеральное агентство по образованию

Южно-Уральский государственный университет

Кафедра прикладной математики

**Курсовая работа**

**по дисциплине «Архитектура ЭВМ и ВС»**

**на тему: «Принципы организации параллелизма выполнения**

**машинных команд в процессорах»**

Выполнила:студентка группы ММ-392

Соловьева М.Н.

Дата «\_\_\_» « »2007 г.

Проверил:

Никитин Г.А.

Дата «\_\_\_» « »2007 г.

Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Челябинск

2007

содержание

введение 3

1 Классификация параллельных ВС 5

1.1 Классификация Флинна 5

1.2 Системы с общей и распределенной памятью 7

2 Конвейеры операций 9

2.1 Конвейеры 9

2.2 Оценка производительности идеального конвейера 10

2.3 Конфликты в конвейере и способы минимизации их влияния на производительность процессора 13

3 Суперскалярные архитектуры 18

3.1 Работа суперскалярного конвейера 18

3.2 Трудности реализации 21

3.3 Историческая справка 22

4 VLIW-архитектура 25

4.1 Аппаратно-программный комплекс VLIW 25

4.2 Устройство VLIW-процессора 26

4.3 Принцип действия VLIW-компилятора 27

4.4 Трудности реализации VLIW 28

5 Предсказание переходов 30

6 Матричные процессоры 35

6.1 Матричные процессоры 35

6.2 Векторный процессор 36

6.3 Внутрипроцессорная многопоточность 37

6.4 Многопоточность в Pentium 4 39

7 Закон Амдала. Закон Густафсона 42

7.1 Ускорение, эффективность, загрузка и качество 42

7.2 Закон Амдала 44

7.3 Закон Густафсона 47

вывод 49

список литературы 50

введение

Спрос на компьютеры, работающие с все более и более высокой скоростью, не прекращается. Астрономы пытаются воспроизвести всю историю Вселенной с момента большого взрыва и до сегодняшнего дня. Фармацевты хотели бы разрабатывать новые лекарственные препараты с помощью компьютеров, не принося в жертву легионы крыс. Разработчики летательных аппаратов могли бы получать лучшие результаты, если бы вместо строительства огромных аэродинамических труб моделировали свои конструкции на компьютере. Какими бы мощными ни были компьютеры, их возможностей никогда не хватит для решения многих нетривиальных задач (особенно научных, технических и промышленных).

Быстродействие процессоров растет, но у них постоянно возникают проблемы со скоростью передачи информации, поскольку скорость распространения электромагнитных волн в медных проводах и света в оптико-волоконных кабелях прежнему остается равной 20 см/нс, независимо от того, насколько умны инженеры компании Intel. Кроме того, чем быстрее работает процессор, тем сильнее он нагревается, поэтому возникает задача защиты его от перегрева.

Разработчики компьютеров стремятся к тому, чтобы повысить производительность своих машин. Один из способов заставить процессоры работать быстрее - повышение их тактовой частоты, однако при этом существуют технологические ограничения. Поэтому большинство разработчиков для повышения производительности при данной тактовой частоте процессора используют параллелизм (выполнение двух или более операций одновременно).

Существует две основные формы параллелизма: параллелизм на уровне команд и параллелизм на уровне процессоров. В первом случае параллелизм реализуется за счет запуска большого количества команд каждую секунду. Во втором случае над одним заданием работают одновременно несколько процессоров. Каждый подход имеет свои преимущества.

Параллелизм можно вводить на разных уровнях. На самом низком уровне он может быть реализован в процессоре за счет конвейеризации и суперскалярной архитектуры с несколькими функциональными блоками.

На следующем уровне возможно внедрение в систему внешних плат ЦП с улучшенными вычислительными возможностями. Как правило, в подключаемых процессорах реализуются специальные функции, такие как обработка сетевых пакетов, обработка мультимедийных данных, криптография. Производительность специализированных приложений за счет этих функций может быть повышена в 5-10 раз.

Чтобы повысить производительность в сто, тысячу или миллион раз, необходимо свести воедино многочисленные процессоры и обеспечить их эффективное взаимодействие. Этот принцип реализуется в виде больших мультипроцессорных систем и мультикомпьютеров (кластерных компьютеров). Естественно, объединение тысяч процессоров в единую систему порождает новые проблемы, которые нужно решать.

Наконец, в последнее время появилась возможность интеграции через Интернет целых организаций. В результате формируются слабо связанные распределенные вычислительные сетки, или решетки. Такие системы только начинают развиваться, но их потенциал весьма высок.

Когда два процессора или обрабатывающих элемента находятся рядом и обмениваются большими объемами данных с небольшими задержками, они называются сильно связанными (tightly coupled). Соответственно, когда два процессора или обрабатывающих элемента располагаются далеко друг от друга и обмениваются небольшими объемами данных с большими задержками, они называются слабо связанными (loosely coupled). [2]

1 Классификация параллельных ВС

1.1 Классификация Флинна

Даже краткое перечисление типов современных параллельных вычислительных систем (ВС) дает понять, что для ориентирования в этом многообразии необходима четкая система классификации. От ответа на главный вопрос — что заложить в основу классификации — зависит, насколько конкретная система классификации помогает разобраться с тем, что представляет собой архитектура ВС и насколько успешно данная архитектура позволяет решать определенный круг задач.

Общепринята удачная классификация ВС, которую предложил в 1970 г. Г. Флин (США). Основным определяющим архитектурным параметром он выбрал взаимодействие потока команд и потока данных (операндов и результатов).

ОКОД — «один поток команд — один поток данных» (SISD - «Single Instruction, Single Data»). В ЭВМ классической архитектуры ведется последовательная обработка команд и данных. Команды поступают одна за другой (за исключением точек ветвления программы), и для них из ОЗУ или регистров также последовательно поступают операнды. Одной команде (операции) соответствует один необходимый ей набор операндов. Представителями этого класса являются, прежде всего, классические фоннеймановские ВМ. То, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка, не имеет значения, поэтому в класс SISD од­новременно попадают как ВМ CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными. Некоторые специалисты считают, что к SISD-системам можно причислить и векторно-конвейерные ВС, если рассматривать вектор как неделимый элемент данных для соответствующей команды.

Тип ОКМД— «один поток команд — много потоков данных» (SIMD — «Single Instruction — Multiple Data») охватывает ВС, в которых одной ко­мандой обрабатывается набор данных, множество данных, вектор, и вы­рабатывается множество результатов. Это векторные и матричные системы, в которых по одной команде выполняется одна и та же операция над всеми элементами массива — вектора или матрицы, распределенными между процессорными (обрабатывающими) элементами ПЭ или процессо­рами. Принцип обработки показан на рисунке 1.2.

Отечественные векторные ВС — ПС-2000, ПС-2100. Допускают организацию матричной обработки. Классический пример матричной архитектуры - ILLIAC-IV (США).

К типу МКОД — «много потоков команд — один поток данных» (MISD — «Multiple Instruction — Single Data») принято относить векторный конвейер (обычно в составе ВС, чтобы подчеркнуть основной используемый принцип вычислений), например, в составе ВС Сгеу-1, «Электроника ССБИС». На векторном конвейере производится последовательная обработка одного потока данных многими обрабатывающими устройствами (ступенями, станциями) конвейера.

К такому же типу относится ВС, реализующая макроконвейер (ВС «Украина»). В ней задача, решаемая циклически, «разрезается» на последовательные этапы, закрепляемые за отдельными процессорами. Запускается конвейер многократного выполнения цикла, составляющего задачу.

Тип МКМД — «много потоков команд — много потоков данных» (MIMD — «Multiple Instruction — Multiple Data»). Класс предполагает на­личие в вычислительной системе множества устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. Класс MIMD чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы. Кроме того, приобщение к классу MIMD зависит от трактовки. Так, ранее упоминавшиеся векторно-конвейерные ВС можно вполне отнести и к классу MIMD, если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) над множественным скалярным потоком.

Схема классификации Флинна вплоть до настоящего времени является наиболее распространенной при первоначальной оценке той или иной ВС, поскольку позволяет сразу оценить базовый принцип работы системы, чего часто бывает достаточно. Однако у классификации Флинна имеются и очевидные недостатки, например неспособность однозначно отнести некоторые архитектуры к тому или иному классу. Другая слабость — это чрезмерная насыщенность класса MIMD. Все это породило множественные попытки либо модифицировать классификацию Флинна, либо предложить иную систему классификации.

1.2 Системы с общей и распределенной памятью

Системы с общей (разделяемой) оперативной памятью образуют со­временный класс ВС — многопроцессорных супер-ЭВМ. Одинаковый доступ всех процессоров к программам и данным представляет широкие возможности организации параллельного вычислительного процесса (параллельных вычислений). Отсутствуют потери реальной производи­тельности на межпроцессорный (между задачами, процессами и т. д.) обмен данными).

Системы с распределенной памятью образуют вычислительные ком­плексы (ВК) — коллективы ЭВМ с межмашинным обменом для совместного решения задач (рис. 1.5б). В ВК объединяются вычислительные средства систем управления, решающие специальные наборы задач, взаимосвязанных по данным. Принято говорить, что такие ВК выполняют распределенные вычисления, а сами ВК называют распределенными ВК.

Другое, противоположное воплощение принципа МИМД — масспроцессорные или высокопараллельные архитектуры, объединяющие сотни — тысячи — десятки тысяч процессоров.

В современных супер-ЭВМ наметилась тенденция объединения двух принципов: общей (распределяемой) и распределенной (локальной) оперативной памяти (ЛОП). Такая структура используется в проекте МВК «Эльбрус-3» и «Эльбрус-ЗМ» .

2 Конвейеры операций

2.1 Конвейеры

Уже много лет известно, что главным препятствием высокой скорости выполнения команд является необходимость их вызова из памяти. Для разрешения этой проблемы можно вызывать команды из памяти заранее и хранить в специальном наборе регистров. Эта идея использовалась еще в 1959 году при разработке компьютера Stretch компании IBM, а набор регистров был назван буфером выборки с упреждением. Таким образом, когда требовалась определенная команда, она вызывалась прямо из буфера, а обращения к памяти не происходило.

В действительности при выборке с упреждением команда обрабатывается за два шага: сначала происходит вызов команды, а затем - ее выполнение. Еще больше продвинула эту стратегию идея конвейера. При использовании конвейера команда обрабатывается уже не за два, а за большее количество шагов, каждый из которых реализуется определенным аппаратным компонентом, причем все эти компоненты могут работать параллельно.

На рисунке 2.1 а) изображен конвейер из пяти блоков, которые называются ступенями. Первая ступень (блок С1) вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не потребуется. Вторая ступень (блок С2) декодирует эту команду, определяя ее тип и тип ее операндов. Третья ступень (блок СЗ) определяет местонахождение операндов и вызывает их из регистров или из памяти. Четвертая ступень (блок С4) выполняет команду, и, наконец, блок С5 записывает результат обратно в нужный регистр.

Чтобы лучше понять принципы работы конвейера, рассмотрим аналогичный пример. Представим себе кондитерскую фабрику, на которой выпечка тортов и их упаковка для отправки производятся раздельно. Предположим, что в отделе отправки находится длинный конвейер, вдоль которого располагаются 5 рабочих (или ступеней обработки). Каждые 10 секунд (это время цикла) первый рабочий ставит пустую коробку для торта на ленту конвейера. Эта коробка отправляется ко второму рабочему, который кладет в нее торт. После этого коробка с тортом доставляется третьему рабочему, который закрывает и запечатывает ее. Затем она поступает к четвертому рабочему, который ставит на ней штамп. Наконец, пятый рабочий снимает коробку с конвейерной ленты и помещает ее в большой контейнер для отправки в супермаркет. Примерно таким же образом действует компьютерный конвейер: каждая команда (в случае с кондитерской фабрикой - торт) перед окончательным выполнением проходит несколько ступеней обработки.

Возвратимся к нашему конвейеру на рисунке 2.1. Предположим, что время цикла у этой машины - 2 нс. Тогда для того, чтобы одна команда прошла через весь конвейер, требуется 10 нс. На первый взгляд может показаться, что такой компьютер будет выполнять 100 млн команд в секунду, в действительности же скорость его работы гораздо выше. В течение каждого цикла (2 нс) завершается выполнение одной новой команды, поэтому машина выполняет не 100, а 500 млн команд в секунду! [2]

2.2 Оценка производительности идеального конвейера

Пусть задана операция, выполнение которой разбито на n последо­вательных этапов. Пусть ti — время выполнения i-го этапа. При последо­вательном их выполнении операция выполняется за время

а быстродействие ЭВМ или одного процессора ВС, выполняющего только эту операцию, составит

Выберем время такта — величину tT = max{ti} и потребуем при раз­биении на этапы, чтобы для любого i = 1,..., n выполнялось условие ti + t(i+1) mod n= tT . То есть чтобы никакие два последовательных этапа (включая конец и новое начало операции) не могли быть выполнены за время одного такта.

Максимальное быстродействие процессора при полной загрузке конвейера составляет

Число n — количество уровней конвейера, или глубина перекрытия, так как каждый такт на конвейере параллельно выполняются n операций. Чем больше число уровней (станций), тем больший выигрыш в быстродействии может быть получен.

Известна оценка

то есть выигрыш в быстродействии получается от до n раз.

Реальный выигрыш в быстродействии оказывается всегда меньше, чем указанный выше, поскольку:

1) некоторые операции, например, над целыми, могут выполняться за меньшее количество этапов, чем другие арифметические операции. Тогда отдельные станции конвейера будут простаивать.

2) при выполнении некоторых операций на определённых этапах могут требоваться результаты более поздних, ещё не выполненных эта­пов предыдущих операций. Приходится приостанавливать конвейер.

3) поток команд порождает недостаточное количество операций для полной загрузки конвейера [3].

Рассмотрим принципы конвейерной обработки информации на примере пятиступенчатого конвейера, в котором выполнение команды складывается из следующих этапов:

IF (Instruction Fetch) - считывание команды в процессор;

ID (Instruction Decoding) - декодирование команды;

OR (Operand Reading) - считывание операндов;

EX (Executing) - выполнение команды;

WB (Write Back) - запись результата.

Выполнение команд в таком конвейере представлено в таблице 2.1.

Так как в каждом такте могут выполняться различные стадии обработки команд, то длительность такта выбирается исходя из максимального времени выполнения всех стадий. Кроме того, следует учитывать, что для передачи команды с одной стадии на другую требуется определенное дополнительное время (Дt), связанное с записью промежуточных результатов обработки в буферные регистры.

|  |
| --- |
| Таблица 2.1  |
| Команда | Такт |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i | IF | ID | OR | EX | WB |  |  |  |  |
| i+1 |  | IF | ID | OR | EX | WB |  |  |  |
| i+2 |  |  | IF | ID | OR | EX | WB |  |  |
| i+3 |  |  |  | IF | ID | OR | EX | WB |  |
| i+4 |  |  |  |  | IF | ID | OR | EX | WB |

Пусть для выполнения отдельных стадий обработки требуются следующие затраты времени (в некоторых условных единицах):

TIF = 20, TID = 15, TOR = 20, TEX = 25, TWB = 20.

Тогда, предполагая, что дополнительные расходы времени составляют Дt = 5 единиц, получим время такта:

T = max {TIF, TID, TOR, TEX, TWB} + Дt = 30.

Оценим время выполнения одной команды и некоторой группы команд при последовательной и конвейерной обработке.

При последовательной обработке время выполнения N команд составит:

Tпосл = N\*(TIF + TID + TOR + TEX + TWB) = 100N.

Анализ таблицы 2.1 показывает, что при конвейерной обработке после того, как получен результат выполнения первой команды, результат очередной команды появляется в следующем такте работы процессора. Следовательно,

Tконв = 5T + (N-1) \* T.

Примеры длительности выполнения некоторого количества команд при последовательной и конвейерной обработке приведены в таблица 2.2.

|  |
| --- |
| Таблица 2.2 |
| Количество команд | Время |
| при последовательном выполнении | при конвейерном выполнении |
| 1 | 100 | 150 |
| 2 | 200 | 240 |
| 10 | 1000 | 420 |
| 100 | 10000 | 3120 |

Очевидно, что при достаточно длительной работе конвейера его быстродействие будет существенно превышать быстродействие, достигаемое при последовательной обработке команд. Это увеличение будет тем больше, чем меньше длительность такта конвейера и чем больше количество выполненных команд. Сокращение длительности такта достигается, в частности, разбиением выполнения команды на большое число этапов, каждый из которых включает в себя относительно простые операции и поэтому может выполняться за короткий промежуток времени. Так, если в процессоре Pentium длина конвейера составляла 5 ступеней (при максимальной тактовой частоте 200 МГц), то в Pentium-4 - уже 20 ступеней (при максимальной тактовой частоте на сегодняшний день 3,4 ГГц).

2.3 Конфликты в конвейере и способы минимизации их влияния на производительность процессора

Значительное преимущество конвейерной обработки перед последовательной имеет место в идеальном конвейере, в котором отсутствуют конфликты и все команды выполняются друг за другом без перезагрузки конвейера. Наличие конфликтов снижает реальную производительность конвейера по сравнению с идеальным случаем.

Конфликты - это такие ситуации в конвейерной обработке, которые препятствуют выполнению очередной команды в предназначенном для нее такте.

Конфликты делятся на три группы:

* структурные,
* по управлению,
* по данным.

Структурные конфликты возникают в том случае, когда аппаратные средства процессора не могут поддерживать все возможные комбинации команд в режиме одновременного выполнения с совмещением.

Причины структурных конфликтов.

1. Не полностью конвейерная структура процессора, при которой некоторые ступени отдельных команд выполняются более одного такта.

Пусть этап выполнения команды i+1 занимает 3 такта. Тогда диаграмма работы конвейера будет иметь вид, представленный в таблица 2.3.

|  |
| --- |
| Таблица 2.3 |
| Команда | Такт |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i | IF | ID | OR | EX | WB |  |  |  |  |
| i+1 |  | IF | ID | OR | EX | EX | EX | WB |  |
| i+2 |  |  | IF | ID | OR | O | O | EX | WB |
| i+3 |  |  |  | IF | ID | OR | O | O | EX |
| i+4 |  |  |  |  | IF | ID | OR | O | O |

При этом в работе конвейера возникают так называемые "пузыри" (обработка команд i+2 и следующих за ней, начиная с такта 6), которые снижают производительность процессора.

Эту ситуацию можно было бы ликвидировать двумя способами. Первый предполагает увеличение времени такта до такой величины, которая позволила бы все этапы любой команды выполнять за один такт. Однако при этом существенно снижается эффект конвейерной обработки, так как все этапы всех команд будут выполняться значительно дольше, в то время как обычно нескольких тактов требует выполнение лишь отдельных этапов очень небольшого количества команд. Второй способ предполагает использование таких аппаратных решений, которые позволили бы значительно снизить затраты времени на выполнение данного этапа (например, использовать матричные схемы умножения). Но это приведет к усложнению схемы процессора и невозможности реализации на этой БИС других функционально более важных узлов. Так как представленная в таблице 2.3 ситуация возникает при реализации команд, относительно редко встречающихся в программе, то обычно разработчики процессоров ищут компромисс между увеличением длительности такта и усложнением того или иного устройства процессора.

2. Недостаточное дублирование некоторых ресурсов.

Одним из типичных примеров служит конфликт из-за доступа к запоминающим устройствам. Из таблицы 2.1 видно, что в случае, когда операнды и команды находятся в одном запоминающем устройстве, начиная с такта 3, работу конвейера придется постоянно приостанавливать, поскольку различные команды в одном и том же такте обращаются к памяти на считывание команды, выборку операнда, запись результата.

Борьба с конфликтами такого рода проводится путем увеличения количества однотипных функциональных устройств, которые могут одновременно выполнять одни и те же или схожие функции. Например, обычно разделяют кэш-память для хранения команд и кэш-память данных, а также используют многопортовую схему доступа к регистровой памяти, при которой к регистрам можно одновременно обращаться по одному каналу для записи, а по другому - для считывания информации. Конфликты из-за исполнительных устройств обычно сглаживаются введением в состав процессора дополнительных блоков.

В суперскалярных процессорах реализована конвейерная обработка и параллельное выполнение команд. Несколько команд одновременно могут выполниться в течение одного такта. В таблице 2.4 представлена последовательность выполнения команд в процессоре, имеющем два конвейера, при условии, что команде К1 требуется 3 такта на этапе EX.

|  |
| --- |
| Таблица 2.4 |
| Этап | Такт |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| IF | K1 | K2 | K3 | K4 | K5 | K6 | K7 | K8 | K7 | K9 | K7 | K10 | K11 | K12 |
| ID |  |  | K1 | K2 | K3 | K4 | K5 | K6 | K5 | K8 | K5 | K9 | K7 | K10 |
| OR |  |  |  |  | K1 | K2 | K3 | K4 | K3 | K6 | K3 | K8 | K5 | K9 |
| EX |  |  |  |  |  |  | K1 | K2 | K1 | K4 | K1 | K6 | K3 | K8 |
| WB |  |  |  |  |  |  |  |  |  | K2 |  | K4 | K1 | K6 |

При этом команды будут завершаться в последовательности К2-К4-К1-К6-...

Следовательно, может нарушиться исходный порядок завершения команд программы. Недостатком суперскалярных процессоров является необходимость синхронного продвижения команд в каждом из конвейеров. При возникновении затора в одном из конвейеров должны приостанавливать свою работу и другие. Но такие приостановки существенно снижают быстродействие процессора. Разрешение этой ситуации состоит в том, чтобы дать возможность выполняться командам в одном конвейере вне зависимости от ситуации в других конвейерах. Это приводит к неупорядоченному выполнению команд. При этом команды, стоящие в программе позже, могут завершиться ранее команд, стоящих впереди. Аппаратные средства процессора должны гарантировать, что результаты выполненных команд будут записаны в приемник в том порядке, в котором команды записаны в программе. Для этого в процессоре результаты этапа выполнения команды обычно сохраняются в специальном буфере восстановления последовательности команд. Запись результата очередной команды из этого буфера в приемник результата проводится лишь после того, как выполнены все предшествующие команды и записаны их результаты.

Конфликты по управлению возникают при конвейеризации команд переходов и других команд, изменяющих значение счетчика команд.

Суть конфликтов этой группы наиболее удобно проиллюстрировать на примере команд условного перехода. Пусть в программе, представленной в таблице 2.1, команда i+1 является командой условного перехода, формирующей адрес следующей команды в зависимости от результата выполнения команды i. Команда i завершит свое выполнение в такте 5. В то же время команда условного перехода уже в такте 3 должна прочитать необходимые ей признаки, чтобы правильно сформировать адрес следующей команды. Если конвейер имеет большую глубину (например, 20 ступеней), то промежуток времени между формированием признака результата и тактом, где он анализируется, может быть еще большим. В инженерных задачах примерно каждая шестая команда является командой условного перехода, поэтому приостановки конвейера при выполнении команд переходов до определения истинного направления перехода существенно скажутся на производительности процессора.

Наиболее эффективным методом снижения потерь от конфликтов по управлению служит предсказание переходов. Суть данного метода заключается в том, что при выполнении команды условного перехода специальный блок процессора определяет наиболее вероятное направление перехода, не дожидаясь формирования признаков, на основании анализа которых этот переход реализуется. Процессор начинает выбирать из памяти и выполнять команды по предсказанной ветви программы (так называемое исполнение по предположению, или "спекулятивное" исполнение). Однако так как направление перехода может быть предсказано неверно, то получаемые результаты с целью обеспечения возможности их аннулирования не записываются в память или регистры (то есть для них не выполняется этап WB), а накапливаются в специальном буфере результатов.

В современных процессорах вероятность правильного предсказания направления переходов достигает 90% [6,11].

Конфликты по данным возникают в случаях, когда выполнение одной команды зависит от результата выполнения предыдущей команды. При обсуждении этих конфликтов будем предполагать, что команда i предшествует команде j[11].

Все виды зависимостей по данным могут быть классифицированы по типу ассо­циаций: RAR — «чтение после чтения», WAR — «запись после чтения» и WAW — «запись после записи», RAW — «чтение после записи».

Некоторые из зависимостей по данным могут быть устранены. RAR, по сути дела, соответствует отсутствию зависимости, поскольку в данном случае порядок выполнения команд не имеет значения. Действительной зависимостью является только «чтение после записи» (RAW), так как необходимо прочитать предварительно записанные новые данные, а не старые.

Лишние зависимости по данным появляются в результате «записи после чтения» (WAR) и «записи после записи» (WAW). Лишние зависимости появляются по нескольким при­чинам: не оптимизированный программный код, ограничение количества регистров, стремление к экономии памяти, наличие программных циклов. Важно отметить, что запись может быть произведена в любой свободный ресурс, а не только тот, который указан в программе[1].

1. Конфликты типа RAW (Read After Write): команда j пытается прочитать операнд прежде, чем команда i запишет на это место свой результат. При этом команда j может получить некорректное старое значение операнда.

Проиллюстрируем этот тип конфликта на примере выполнения команд, представленных в таблице 2.1. Пусть выполняемые команды имеют следующий вид:

|  |  |  |
| --- | --- | --- |
| i | ADD R1,R2  | R1 = R1+R2 |
| i+1=j  | SUB R3,R1  | R3 = R3-R1 |

Команда i изменит состояние регистра R1 в такте 5. Но команда i+1 должна прочитать значение операнда R1 в такте 4. Если не приняты специальные меры, то из регистра R1 будет прочитано значение, которое было в нем до выполнения команды i.

Уменьшение влияния конфликта типа RAW обеспечивается методом обхода (продвижения) данных. В этом случае результаты, полученные на выходах исполнительных устройств, помимо входов приемника результата передаются также на входы всех исполнительных устройств процессора. Если устройство управления обнаруживает, что данный результат требуется одной из последующих команд в качестве операнда, то он сразу же, параллельно с записью в приемник результата, передается на вход исполнительного устройства для использования следующей командой.

Конфликты типа RAW обусловлены именно конвейерной организацией обработки команд.

Главной причиной двух других типов конфликтов по данным является возможность неупорядоченного выполнения команд в современных роцессорах, то есть выполнение команд не в том порядке, в котором они записаны в программе.

2. Конфликты типа WAR (Write After Read): команда j пытается записать результат в приемник, прежде чем он считается оттуда командой i, При этом команда i может получить некорректное новое значение операнда:

Этот конфликт возникнет в случае, если команда j вследствие неупорядоченного выполнения завершится раньше, чем команда i прочитает старое содержимое регистра R2.

3. Конфликты типа WAW (Write After Write): команда j пытается записать результат в приемник, прежде чем в этот же приемник будет записан результат выполнения команды i, то есть запись заканчивается в неверном порядке, оставляя в приемнике результата значение, записанное командой i:

Устранение конфликтов по данным типов WAR и WAW достигается путем отказа от неупорядоченного исполнения команд, но чаще всего путем введения буфера восстановления последовательности команд.

Как отмечалось выше, наличие конфликтов приводит к значительному снижению производительности процессора. Определенные типы конфликтов требуют приостановки конвейера. При этом останавливается выполнение всех команд, находящихся на различных стадиях обработки. Другие конфликты при неверном предсказанном направлении перехода, ведут к необходимости полной перезагрузки конвейера. Потери будут тем больше, чем более длинный конвейер используется в процессоре. Такая ситуация явилась одной из причин сокращения числа ступеней в процессорах последних моделей [11].

3 Суперскалярные архитектуры

3.1 Работа суперскалярного конвейера

Одна из возможных схем процессора с двумя конвейерами показана на рисунке 3.1. В ее основе лежит конвейер, изображенный на рисунке 2.1. Здесь общий блок выборки команд вызывает из памяти сразу по две команды и помещает каждую из них в один из конвейеров. Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать из-за ресурсов (например, регистров), и ни одна из них не должна зависеть от результата выполнения другой. Как и в случае с одним конвейером, либо компилятор должен гарантировать отсутствие нештатных ситуаций (когда, например, аппаратура не обеспечивает проверку команд на несовместимость и при обработке таких команд выдает некорректный результат), либо за счет дополнительной аппаратуры конфликты должны выявляться и устраняться непосредственно в ходе выполнения команд.

Сначала конвейеры (как сдвоенные, так и обычные) использовались только в RISC-компьютерах. У процессора 386 и его предшественников их не было. Конвейеры в процессорах компании Intel появились, только начиная с модели 486. Процессор 486 имел один пятиступенчатый конвейер, a Pentium - два таких конвейера. Похожая схема изображена на рисунке 3.1, но разделение функций между второй и третьей ступенями (они назывались декодер 1 и декодер 2) было другим. Главный конвейер (u-конвейер) мог выполнять произвольные команды. Второй конвейер (v-конвейер) мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH) [2,5].

Имеются сложные правила определения, является ли пара команд совместимой в отношении возможности параллельного выполнения. Если команды, входящие в пару, были сложными или несовместимыми, выполнялась только одна из них (в u-конвейере). Оставшаяся вторая команда составляла затем пару со следующей командой. Команды всегда выполнялись по порядку. Таким образом, процессор Pentium содержал особые компиляторы, которые объединяли совместимые команды в пары и могли порождать программы, выполняющиеся быстрее, чем в предыдущих версиях. Измерения показали, что программы, в которых применяются операции с целыми числами, при той же тактовой частоте на Pentium выполняются почти в два раза быстрее, чем на 486. Вне всяких сомнений, преимущество в скорости было достигнуто благодаря второму конвейеру.

Стоит отметить, что переход к четырем конвейерам возможен, но требует громоздкого аппаратного обеспечения. Вместо этого используется другой подход. Основная идея - один конвейер с большим количеством функциональных блоков, как показано на рисунке 3.2. Pentium II, к примеру, имеет сходную структуру. В 1987 году для обозначения этого подхода был введен термин суперскалярная архитектура. Однако подобная идея нашла воплощение еще тридцатью годами ранее в компьютере CDC 6600. Этот компьютер вызывал команду из памяти каждые 100 не и помещал ее в один из 10 функциональных блоков для параллельного выполнения. Пока команды выполнялись, центральный процессор вызывал следующую команду.

Со временем значение понятия "суперскалярный" несколько изменилось. Теперь суперскалярными называют процессоры, способные запускать несколько команд зачастую от четырех до шести) за один тактовый цикл. Естественно, чтобы передавать все эти команды, в суперскалярном процессоре должно быть несколько функциональных блоков. Поскольку в процессорах этого типа, как правило, предусматривается один конвейер, его устройство обычно соответствует рисунку 3.2.

В свете такой терминологической динамики на сегодняшний день можно утверждать, что компьютер 6600 не был суперскалярным с технической точки зрения - ведь за один тактовый цикл в нем запускалось не больше одной команды. Однако при этом был достигнут аналогичный результат - команды запускались быстрее, чем выполнялись. На самом деле разница в производительности между ЦП с циклом в 100 не, передающим за этот период по одной команде четырем функциональным блокам, и ЦП с циклом в 400 не, запускающим за это время четыре команды, трудноуловима. В обоих процессорах соблюдается принцип превышения скорости запуска над скоростью управления; при этом рабочая нагрузка распределяется между несколькими функциональными блоками.

Отметим, что на выходе ступени 3 команды появляются значительно быстрее, чем ступень 4 способна их обрабатывать. Если бы на выходе ступени 3 команды появлялись каждые 10 не, а все функциональные блоки делали свою работу также за 10 не, то на ступени 4 всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной. Как видно из рисунка 3.2, на ступени 4 может быть несколько АЛУ.

Суперскалярные процессоры имеют:

* многоуровневую иерархическую память, включая до трех уровней кэш-памя­ти;
* раздельные кэш-памяти команд и данных;
* устройство выборки команд, обеспечивающее выборку на исполнение сово­купности команд;
* таблицы предсказания переходов;
* переименование регистров;
* поддержку внеочередного исполнения команд;
* набор функциональных устройств для преобразования данных в форматах с фиксированной и плавающей точкой.

Суперскалярные машины используют параллелизм на уровне команд путем посылки нескольких команд из обычного потока команд в несколько функциональных устройств. Дополнительно, чтобы снять ограничения последовательного выполнения команд, эти машины используют механизмы внеочередной выдачи и внеочередного завершения команд, прогнозирование переходов, кэши целевых адресов переходов и условное (по предположению) выполнение команд. Возросшая сложность, реализуемая этими механизмами, создает также проблемы реализации точного прерывания.

В типичной суперскалярной машине аппаратура может осуществлять выдачу от одной до шести команд в одном такте. Обычно эти команды должны быть независимыми и удовлетворять некоторым ограничениям, например таким, что в каждом такте не может выдаваться более одной команды обращения к памяти. Если какая-либо команда в потоке команд является логически зависимой или не удовлетворяет критериям выдачи, на выполнение будут выданы только команды, предшествующие данной. Поэтому скорость выдачи команд в суперскалярных машинах является переменной. Это отличает их от VLIW-машин, в которых полную ответственность за формирование пакета команд, которые могут выдаваться одновременно, несет компилятор, а аппаратура в динамике не принимает никаких решений относительно выдачи нескольких команд.

Предположим, что машина может выдавать на выполнение две команды в одном такте. Одной из таких команд может быть команда загрузки регистров из памяти, записи регистров в память, команда переходов, операции целочисленного АЛУ, а другой может быть любая операция плавающей точки (ПТ). Параллельная выдача целочисленной операции и операции с плавающей точкой намного проще, чем выдача двух произвольных команд. В реальных системах (например, в процессорах PA7100, hyperSPARC, Pentium и др.) применяется именно такой подход. В более мощных процессорах (например, MIPS R10000, UltraSPARC, PowerPC 620 и др.) реализована выдача до четырех команд в одном такте.

Выдача двух команд в каждом такте требует одновременной выборки и декодирования по крайней мере 64 бит. Чтобы упростить декодирование можно потребовать, чтобы команды располагались в памяти парами и были выровнены по 64-битовым границам. В противном случае необходимо анализировать команды в процессе выборки и, возможно, менять их местами в момент пересылки в целочисленное устройство и в устройство ПТ. При этом возникают дополнительные требования к схемам обнаружения конфликтов. В любом случае вторая команда может выдаваться, только если может быть выдана на выполнение первая команда. Аппаратура принимает такие решения в динамике, обеспечивая выдачу только первой команды, если условия для одновременной выдачи двух команд не соблюдаются. В таблице 3.1 представлена диаграмма работы подобного конвейера в идеальном случае, когда в каждом такте на выполнение выдается пара команд.

Такой конвейер позволяет существенно увеличить скорость выдачи команд. Однако чтобы он смог так работать, необходимо иметь либо полностью конвейеризованные устройства плавающей точки, либо соответствующее число независимых функциональных устройств. В противном случае устройство плавающей точки станет узким горлом и эффект, достигнутый за счет выдачи в каждом такте пары команд, сведется к минимуму.

Рассмотрим следующие этапы выполнения команды:

* выборка команды - IF;
* декодирование команды - ID;
* выполнение операции - EX;
* обращение к памяти - MEM;
* запоминание результата - WB.

|  |  |
| --- | --- |
| Тип команды | Ступень конвейера  |
| Целочисленная команда | IF | ID | EX | MEM | WB |  |  |  |
| Команда ПТ  | IF | ID | EX | MEM | WB |  |  |  |
| Целочисленная команда |  | IF | ID | EX | MEM | WB |  |  |
| КомандаПТ |  | IF | ID | EX | MEM | WB |  |  |
| Целочисленная команда |  |  | IF | ID | EX | MEM | WB |  |
| Команда ПТ |  |  | IF | ID | EX | MEM | WB |  |
| Целочисленная команда |  |  |  | IF | ID | EX | MEM | WB |
| Команда ПТ |  |  |  | IF | ID | EX | MEM | WB |

Таблица 3.1 Работа суперскалярного конвейера

3.2 Трудности реализации

При параллельной выдаче двух операций (одной целочисленной команды и одной команды ПТ) потребность в дополнительной аппаратуре, помимо обычной логики обнаружения конфликтов, минимальна: целочисленные операции и операции ПТ используют разные наборы регистров и разные функциональные устройства. Единственная сложность возникает, только если команды представляют собой команды загрузки, записи и пересылки чисел с плавающей точкой. Эти команды создают конфликты по портам регистров ПТ, а также могут приводить к новым конфликтам типа RAW, когда операция ПТ, которая могла бы быть выдана в том же такте, является зависимой от первой команды в паре.

Если пара команд состоит из одной команды загрузки с ПТ и одной операции с ПТ, которая от нее зависит, необходимо обнаруживать подобный конфликт и блокировать выдачу операции с ПТ. За исключением этого случая, все другие конфликты естественно могут возникать, как и в обычной машине, обеспечивающей выдачу одной команды в каждом такте. Для предотвращения ненужных приостановок могут, правда, потребоваться дополнительные цепи обхода.

Другой проблемой, которая может ограничить эффективность суперскалярной обработки, является задержка загрузки данных из памяти. В нашем примере простого конвейера команды загрузки имели задержку в один такт, что не позволяло следующей команде воспользоваться результатом команды загрузки без приостановки. В суперскалярном конвейере результат команды загрузки не может быть использован в том же самом и в следующем такте. Это означает, что следующие три команды не могут использовать результат команды загрузки без приостановки. Задержка перехода также становится длиною в три команды, поскольку команда перехода должна быть первой в паре команд. Чтобы эффективно использовать параллелизм, доступный на суперскалярной машине, нужны более сложные методы планирования потока команд, используемые компилятором или аппаратными средствами, а также более сложные схемы декодирования команд.

В общем случае в суперскалярной системе команды могут выполняться параллельно и возможно не в порядке, предписанном программой. Если не предпринимать никаких мер, такое неупорядоченное выполнение команд и наличие множества функциональных устройств с разными временами выполнения операций могут приводить к дополнительным трудностям. Например, при выполнении некоторой длинной команды с плавающей точкой (команды деления или вычисления квадратного корня) может возникнуть исключительная ситуация уже после того, как завершилось выполнение более быстрой операции, выданной после этой длинной команды. Для того, чтобы поддерживать модель точных прерываний, аппаратура должна гарантировать корректное состояние процессора при прерывании для организации последующего возврата.

Обычно в машинах с неупорядоченным выполнением команд предусматриваются дополнительные буферные схемы, гарантирующие завершение выполнения команд в строгом порядке, предписанном программой. Такие схемы представляют собой некоторый буфер "истории", то есть аппаратную очередь, в которую при выдаче попадают команды и текущие значения регистров результата этих команд в заданном программой порядке.

3.3 Историческая справка

В 1993 году корпорация Intel внедрила в массовое производство параллелизм на уровне команд, выпустив процессор Intel Pentium, обладавший способностью декодировать и выполнять команды вычислительного потока параллельно. Годом позже специалисты Intel реализовали двухпроцессорную обработку (два полноценных процессора помещались в два разъема на одной системной плате), создав аппаратную многопоточную среду для серверов и рабочих станций. В 1995 году был представлен процессор Intel Pentium Pro, поддерживавший эффективное объединение четырех процессоров на одной системной плате, что позволило обеспечить более высокую скорость обработки данных в многопоточных приложениях, ориентированных на серверные платформы и рабочие станции.

Появление в 2002 году технологии Hyper-Threading (HT) ознаменовало приход многопоточного параллелизма, то есть возможности выполнять разные потоки приложений одновременно на одноядерном процессоре. Тестирование производительности, проведенное корпорацией Intel, показало, что на процессорах с технологией HT скорость работы некоторых приложений возрастает в среднем на 30%.

Ныне, взяв курс на многоядерные платформы, корпорация Intel стала лидером в процессе перехода на многопоточные и параллельные вычисления на массовых ПК, обеспечив обработку данных на нескольких вычислительных ядрах одного процессора.

Большинство приложений, уже сегодня оптимизированных для параллельного исполнения вычислительных потоков, например, программ, поддерживающих технологию Hyper-Threading или предназначенных к исполнению на рабочих станциях или серверах с двухпроцессорной конфигурацией, при выполнении на многоядерном процессоре демонстрируют прекрасную масштабируемость производительности. К этой категории относятся мультимедийные приложения, научные приложения и системы CAD/CAM [7,9].

Первый суперскалярный МП i960 был выпущен фирмой Intel в 1987 году. Затем были разработаны МП SPARC (1987-1989 годы), MIPS (1988-1989 годы), МПi860 (1989 год)и ряд других суперскалярных МП, в частности:

1. Процессор Pentium был впервые поставлен фирмой Intel в 1993 году как продолжение семейства МП 80x86. Цель его создания - получение быстродействия RISC-МП и полная совместимость на уровне двоичных кодов с программным обеспечением, созданным для всех МП 80x86.
2. Группа фирм AIM (APPLE + IBM + MOTOROLA) совместно разработали семейство МП POWER PC и выпустили его первый образец МП 661 в 1993 году.
3. Фирма DEC в 1992 году для создания мощных рабочих станций выпустила МП 21064 с тактовой частотой 250 Мгц, а затем более мощный МП - 21164.
4. В 1994 году фирма MIPS Computer, известная разработкой суперконвейерных МП, выпустила первый суперскалярный МП MIPS R8000 (MIPS - Microprocessor Without Interlocked Pipeline Stages), а затем МП R10000.
5. В 1994 году фирма Sun Microsystem Inc. в продолжение развития своей серии SPARC (Scalable Processor Architecture) выпустила мощный МП UltraSPARC.
6. В 1994-1995 годах фирмой Hewlett-Packard был выпущен МП PA7200 с высокими показателями быстродействия, предполагается к выпуску МП РА8000.

Все указанные МП являются суперскалярными и поэтому характеризуются рядом общих свойств, в частности:

1. Формирование группы команд для загрузки конвейеров производится динамически в каждом такте. Для этого аппаратно на этапе предвыборки и дешифрации производится анализ зависимости по данным смежных команд. В конвейеры для параллельного исполнения подбираются независимые команды, при этом допускается изменение порядка выполнения команд.
2. Все МП используют динамическое прогнозирование ветвлений на основе буфера истории переходов. Иногда используется одновременное выполнение альтернативных ветвей.
3. Некоторые МП строятся таким образом, что число физических регистров превышает число РОН, определенных архитектурно (РРС620, Mips R10000, P6). Это необходимо для реализации альтернативных ветвей при переходах и для устранения зависимостей по данным, вызванных недостатком РОН. В процессе выполнения команд необходимо производить переименование физических регистров, то есть они выступают в качестве виртуальных.

Большинство указанных МП выпускается в однокристальном исполнении, однако в целях получения более высокого быстродействия для МП PPC 620 использовано 10 кристаллов пяти типов, а для МП R8000 - 4 кристалла трех типов.

Архитектура описанных выше суперскалярных МП приобретает традиционный характер, поэтому предпринимаются попытки освоить новые архитектуры. Одной из наиболее перспективных является разработка МП РА9000, производимая совместно фирмами Hewlett-Packard и Intel. Главная особенность РА9000 состоит в том , что генерация набора команд для одного такта полностью переносится в компилятор, что позволяет достичь высокого уровня оптимальности программы и значительно разгрузить кристалл от схем планирования и упаковки. Тем самым совершается переход к VLIW (Very Long Instruction Word) архитектуре [8,10].

4 VLIW-архитектура

В 1970 г. многие вычислительные системы оснащались дополнительными векторными сигнальными процессорами (VSP - Vector Signal Processor), использующими VLIW-подобные длинные инструкции, прошитые в ПЗУ. Эти процессоры применялись для выполнения быстрого преобразования Фурье (БПФ) и других вычислительных алгоритмов.

Первыми настоящими VLIW-компьютерами стали мини-суперкомпьютеры, выпущенные в начале 1980 года компаниями MultiFlow, Culler и Cydrome, но они не имели коммерческого успеха. Планировщик вычислений и программная конвейеризация были предложены Фишером и Рау (Cydrome). Сегодня это является основой технологии VLIW-компилятора.

Первый VLIW-компилятор компании Multi-Flow 7/300 использовал два АЛУ для целых чисел, два АЛУ для чисел с плавающей точкой и блок логического ветвления. Все это было собрано на нескольких микросхемах. Его 256-битное слово инструкции содержало семь 32-битных кодов операций. Модули для обработки целых чисел могли выполнять 2 операции за один такт длиной 130 нс (то есть всего 4 при двух АЛУ), что при обработке целых чисел обеспечивало быстродействие около 30MIPS (Million Instruction Per Second). Первый VLIW-компьютер Cydrome Cydra-5 использовал 256-битную инструкцию и специальный режим, обеспечивающий выполнение инструкций как последовательности из шести 40-битных операций. Поэтому его компиляторы могли генерировать смесь параллельного кода и обычного последовательного. Существует мнение, что в то время, как эти VLIW-машины использовали несколько микросхем, процессор Intel i860 стал первым VLIW-процессором на одной микросхеме. При установке правильной последовательности операций этот процессор в большей степени зависит от компилятора, нежели от аппаратуры.

Несмотря на то, что архитектура VLIW появилась еще на заре компьютерной индустрии (Тьюринг разработал VLIW-компьютер еще в 1946 году), она до сих пор не имела коммерческого успеха. Однако значительного повышения производительности и скорости вычислений можно добиться лишь путем переноса интеллектуальных функций из аппаратного обеспечения в программное (в компилятор). В целом успех этого мероприятия будет определяться в основном программными средствами, именно в этом и состоит проблема.

4.1 Аппаратно-программный комплекс VLIW

Архитектура VLIW представляет собой одну из последних реализаций концепции внутреннего параллелизма в процессорах. Их быстродействие можно повысить двумя способами: увеличив либо тактовую частоту, либо количество операций, выполняемых за один такт. В первом случае требуется изобретение "быстрых" технологий (например, использование арсенида галлия или кремния на сапфире) и применение таких архитектурных решений, как глубинная конвейеризация (конвейеризация в пределах одного такта, когда в каждый момент времени задействован весь кристалл, а не отдельные его части). Для увеличения количества выполняемых за один цикл операций необходимо на одной микросхеме разместить множество функциональных модулей обработки и обеспечить надежное параллельное исполнение машинных инструкций, что дает возможность включить в работу все модули одновременно. Надежность в таком контексте означает, что результаты вычислений будут правильными. Для примера рассмотрим два выражения, которые связаны друг с другом следующим образом: А=В+С и В=D+Е. Значение переменной А будет разным в зависимости от порядка, в котором вычисляются эти выражения (сначала А, а потом В, или наоборот), но в программе подразумевается только одно определенное значение.

Планирование порядка вычислений довольно трудная задача, которую приходится решать при проектировании современного процессора. В суперскалярных процессорах (процессор с двумя и более конвейерами, что позволяет выполнять более одной команды за один такт в идеальных условиях) для распознавания зависимостей между машинными инструкциями применяется специальное довольно сложное аппаратное решение (в процессоре Pentium Pro, например, для этого используется буфер переупорядочивания инструкций). Однако размеры такого аппаратного планировщика при увеличении количества функциональных модулей обработки возрастают в геометрической прогрессии, что, в конце концов, может "съесть" весь кристалл процессора. Поэтому суперскалярные проекты остановились на отметке пять-шесть управляемых за цикл инструкций. При другом подходе можно передать все планирование программному обеспечению, как это делается в конструкциях с VLIW. "Умный" компилятор должен выискать в программе все инструкции, которые являются совершенно независимыми, собрать их вместе в очень длинные строки (длинные инструкции) и затем отправить на одновременное исполнение функциональными модулями, количество которых строго равно количеству операций в такой длинной инструкции. Очень длинные инструкции обычно имеют размер от 256 бит до 1024 бит. Размер полей, кодирующих операции для каждого функционального модуля, в такой метаинструкции намного меньше.

4.2 Устройство VLIW-процессора

Процессор VLIW, имеющий такую схему, может выполнять восемь операций за один такт и работать при аналогичной тактовой частоте на 80-100% быстрее существующих суперскалярных чипов. Добавочные функциональные блоки могут повысить производительность (за счет уменьшения конфликтов), не слишком усложняя чип. Однако это расширение ограничивается физическими возможностями: количеством портов чтения-записи, необходимым для обеспечения одновременного доступа функциональных блоков к файлу, регистров и взаимосвязей, которое геометрически растет при увеличении количества функциональных блоков. К тому же компилятор должен распараллелить программу до необходимого уровня, чтобы обеспечить загрузку каждому блоку. Процессор выполняет 8 операций за один цикл.

Эта гипотетическая инструкция длиной в 256 бит имеет восемь операционных полей, каждое из которых выполняет традиционную трехоперандную инструкцию (< оп. > < рег. источник > < рег. получатель >). Каждое операционное поле может непосредственно управлять специфическим функциональным блоком при минимальном декодировании.

Аппаратная реализация VLIW-процессора очень проста: несколько небольших функциональных модулей (сложения, умножения, ветвления и т.д.), подключенных к шине процессора, и несколько регистров и блоков кэш-памяти. VLIW-архитектура представляет интерес для полупроводниковой промышленности по двум причинам. Первая причина - теперь на кристалле больше места может быть отведено для блоков обработки, а не, скажем, для блока предсказания переходов. Вторая причина - VLIW-процессор может быть высокоскоростным, так как предельная скорость обработки определяется только внутренними особенностями самих функциональных модулей.

VLIW изымает микрокод из процессора и переносит его в компилятор, в результате чего эмуляция инструкций процессора 8086, таких, как STOS, осуществляется очень эффективно, поскольку процессор получает для исполнения уже готовые макросы. Но вместе с тем это порождает и некоторые трудности, ведь написание микрокода - невероятно трудоемкий и длительный процесс. Архитектуре VLIW может обеспечить жизнеспособность только "умный" компилятор, который возьмет эту работу на себя. Именно это ограничивает использование вычислительных машин с архитектурой VLIW: пока она нашла свое применение только в векторных (для научных расчетов) и сигнальных процессорах.

4.3 Принцип действия VLIW-компилятора

Вновь вспыхнувший в последнее время интерес к VLIW, как к архитектуре, которую можно использовать для реализации вычислений общего назначения, дал существенный толчок развитию техники компиляции для VLIW. VLIW-компилятор упаковывает группы независимых операций в очень длинные слова инструкций таким способом, чтобы обеспечить эффективное их исполнение функциональными модулями за один машинный такт. Компилятор сначала обнаруживает все зависимости между данными, а затем определяет, как их развязать. Чаще всего это делается путем переупорядочивания всей программы, разные ее блоки перемещаются с одного места в другое. Этот подход отличается от применяемого в суперскалярном процессоре, который для определения зависимостей использует специальное аппаратное решение прямо во время выполнения программы (оптимизирующие компиляторы, безусловно, улучшают работу суперскалярного процессора, но не делают его "привязанным" к ним). Большинство суперскалярных процессоров может обнаружить зависимости и планировать параллельное исполнение только внутри базовых программных блоков (группа последовательных операторов программы, не содержащих внутри себя останова или логического ветвления, допустимых только в конце).

Для обеспечения большего параллелизма VLIW-компьютеры должны наблюдать за операциями из разных базовых блоков, чтобы поместить эти операции в одну и ту же длинную инструкцию, их "область обзора" должна быть шире, чем у суперскалярных процессоров. Это обеспечивается путем прокладки "маршрута" по всей программе (трассировка). Трассировка - наиболее оптимальный для некоторого набора исходных данных маршрут по программе для обеспечения правильного результата, гарантирует непересечение этих данных. То есть маршрут, который "проходит" по участкам, пригодным для параллельного выполнения (эти участки формируются, кроме всего прочего, и путем переноса кода из других мест программы), после чего остается упаковать эти участки в длинные инструкции и передать на выполнение. Планировщик вычислений осуществляет оптимизацию на уровне всей программы, а не ее отдельных базовых блоков. Для VLIW, так же, как и для RISC, ветвления в программе являются "врагом", препятствующим эффективному ее выполнению: типичный программный код (тот, что не используется в научных расчетах) содержит около шести ветвлений на инструкцию. В то время как RISC для прогнозирования ветвлений использует аппаратное решение, VLIW оставляет это за компилятором. Компилятор использует информацию, собранную им путем профилирования программы, хотя у будущих VLIW-процессоров предполагается небольшое аппаратное расширение, обеспечивающее сбор для компилятора статистической информации непосредственно во время выполнения программы. Компилятор прогнозирует наиболее подходящий маршрут и планирует его прохождение, рассматривая его как один большой базовый блок, а затем повторяет этот процесс для всех других возникших после этого программных веток, и так до самого конца программы. Он также умеет делать при анализе кода и другие "умные шаги", такие, как развертывание программного цикла и IF-преобразование, в процессе которого временно удаляются все логические переходы из секции, подвергающейся трассировке. Там, где RISC может только просмотреть код вперед на предмет ветвлений, VLIW-компилятор перемещает его с одного места в другое до обнаруженного ветвления (согласно трассировке), но предусматривает при необходимости возможность отката назад, к предыдущему программному состоянию. Соответствующее аппаратное обеспечение, добавленное к VLIW-процессору, может оказать определенную поддержку компилятору. Например, операции, имеющие по несколько ветвлений, могут входить в одну длинную инструкцию и, следовательно, выполняться за один машинный такт. Поэтому выполнение условных операций, которые зависят от результатов предыдущих, может быть реализовано программным способом, а не аппаратным. Цена, которую приходится платить за увеличение быстродействия VLIW-процессора, намного меньше стоимости компиляции. Именно поэтому основные расходы приходятся на компиляторы.

4.4 Трудности реализации VLIW

При реализации архитектуры VLIW возникают и другие серьезные проблемы: VLIW-компилятор должен в деталях "знать" внутренние особенности архитектуры процессора, опускаясь до внутреннего устройства самих функциональных модулей. Как следствие, при выпуске новой версии VLIW-процессора с большим количеством обрабатывающих модулей (или даже с тем же количеством, но другим быстродействием) все старое программное обеспечение, скорее всего, потребует полной перекомпиляции. Надо ли было при переходе, скажем, на процессор 486 избавляться от имеющегося ПО для процессора 386? Конечно, нет, а вот при переходе от одного VLIW-процессора к другому придется, и это разработчик должен учесть при планировании своих затрат - потребуются дополнительные средства на перекомпиляцию. Сторонники VLIW-архитектуры в оправдание предлагают разделить процесс компиляции на две стадии. Все программное обеспечение должно готовиться в аппаратно-независимом формате с использованием промежуточного кода, который окончательно транслируется в машинно-зависимый код только после установки на машине пользователя. Пример такого подхода демонстрирует фонд OSF со своим стандартом ANDF (Architecture-Neutral Distribution Format). Но кросс-платформенное программное обеспечение пока еще только желаемое, а в действительности разработчики ПО для ПК зачастую весьма инертны по отношению к принятию радикально новых технологий. Другая трудность - это по своей сути статическая природа оптимизации, которую обеспечивает VLIW-компилятор. Как поведет себя программа, когда столкнется во время компиляции с непредусмотренными динамическими ситуациями, такими как, например, ожидание ввода-вывода? Архитектура VLIW возникла в ответ на требования со стороны научно-технических организаций, где при вычислениях особенно необходимо большое быстродействие процессора, но для объектно-ориентированных и управляемых по событиям программ она менее подходит, а ведь именно такие программы составляют сейчас большинство в мире ПК. Но и это еще не все: а как можно проверить, что компилятор выполняет такие сложные преобразования надежно и правильно? Пока никак. Вот почему VLIW-компиляторы называют "вещью в себе". Однако решение сложной задачи обеспечения взаимодействия аппаратного и программного обеспечения в архитектуре VLIW требует серьезных предварительных исследований.

Таким образом, достоинства VLIW заключаются в следующем. Во-первых, компилятор может более эффективно исследовать зависимости между командами и выбирать параллельно исполняемые команды, чем это делает аппаратура суперскалярного процессора, ограниченная размером окна исполнения.

Во-вторых, VLIW процессор имеет более простое устройство управления и по­тенциально может иметь более высокую тактовую частоту.

Однако у VLIW процессоров есть серьезный фактор, снижающий их произво­дительность. Это команды ветвления, зависящие от данных, значения которых ста­новятся известны только в динамике вычислений. Окно исполнения VLIW-процессора не может быть очень большим ввиду отсутствия у компилятора информации о зависимостях, формируемых динамически, в процессе выполнения. Этот недостаток препятствует возможности переупорядочивания операций в VLIW процессор. Кроме того, VLIW реализация требует большого размера памяти имен, многовходовых регистровых файлов, большого числа перекрестных связей. Возможен также останов, когда во время выполнения возникла ситуация, отличающаяся от состояния в момент генерации плана выполнения (например, во время выполнения произошло неудачное обращение в кэш-память).

5 Предсказание переходов

Команды, помещенные в окно исполнения, могут быть зависимы по данным. Эти зависимости обусловлены использованием одних и тех же ресурсов памяти (регистров, ячеек памяти) в разных командах. Поэтому для правильного исполнения программы необходимо использование этих ресурсов в предписываемом программой порядке.

Поскольку при суперскалярной обработке необходимо извлекать из памяти не­сколько команд за один такт для загрузки параллельно работающих функциональных устройств, повышенные требования предъявляются к пропускной способности интерфейса «процессор-память». В современных процессорах применяются многоуровневые раздельные кэш-памяти данных и команд.

Для уменьшения потерь процессорных тактов, связанных с промахами при обра­щении к кэш-памяти в случае выполнения команд ветвления, в состав системы кэширования вводятся средства предсказания переходов, основное назначение которых — повысить вероятность наличия в кэшпамяти требуемой команды.

Исполнение условных ветвлений состоит из следующих этапов:

* распознавание команды условного ветвления;
* проверка выполнения условия перехода;
* вычисление адреса перехода;
* передача управления в случае перехода.

На каждом этапе используются специальные приемы повышения производи­тельности [1].

1. Для быстрого декодирования применяются либо дополнительные биты в поле команды, либо преддекодирование команд при их выборке из кэш-памяти команд.

2. Часто, когда команда уже выбрана из кэш-памяти команд, условие перехода еще не вычислено. Чтобы не задерживать поток команд, в данном случае используется предсказание перехода по одной из нескольких возможных схем.

Механизм предсказания переходов выполняет две основные функции — предсказание программного адреса инструкции, на которую производится переход (для всех инструкций перехода), и предсказание направления ветвления (для инструкций условного перехода). Оба предсказания должны быть выполнены заблаговременно — раньше, чем начнётся декодирование и обработка инструкции перехода — для того, чтобы выборка нового блока инструкций была произведена без потерь лишних тактов либо с минимальными потерями.

Необходимость предсказания адреса «целевой» инструкции вызвана тем, что этот адрес может быть извлечён из x86-инструкции перехода и вычислен только на финальной стадии декодирования, с большой задержкой. Более того, даже простое выделение инструкций переменной длины из считанного блока и поиск среди них инструкций перехода займёт какое-то время. Поэтому в процессорах архитектуры x86 предсказание производят по целому блоку, без разбиения его на составляющие инструкции.

В современных процессорах для предсказания адреса перехода обычно используют специальную таблицу адресов переходов BTB (Branch Target Buffer). Эта таблица устроена подобно кэшу и содержит адреса инструкций, на которые ранее производились переходы. Например, в процессоре P-III таблица BTB имеет размер 512 элементов и организована в виде 128 наборов с ассоциативностью 4. Для адресации набора используются младшие разряды адреса 16-байтового блока инструкций. Если в этом блоке есть инструкции перехода, и если эти инструкции отрабатывали ранее, то алгоритм предсказания может очень быстро найти адрес целевой инструкции в таблице BTB и начать считывание блока, содержащего эту инструкцию. Адреса целевых инструкций помещаются в BTB в момент отставки соответствующих инструкций перехода.

В других современных процессорах размер таблицы BTB достигает 2048 элементов (K8) и 4096 элементов (P-4). Организация данной подсистемы в процессоре K8 несколько отличается от классической и основывается на предварительной разметке блоков инструкций в так называемых массивах селекторов перед помещением их в I-кэш. Эти селекторы привязаны к положению инструкций в I-кэше и при их вытеснении оттуда сохраняются в L2-кэше (в так называемых ECC-битах, предназначающихся для коррекции ошибок). Элементы таблицы BTB также привязаны к положению инструкций в I-кэше и теряются при их вытеснении. Это несколько снижает эффективность предсказания адресов переходов в процессоре K8.

Для предсказания направления условного перехода используется другой механизм, основанный на изучении поведения переходов в программе в процессе её выполнения (своего рода «сбор статистики»). Этот механизм учитывает как локальное поведение конкретной инструкции перехода (например, «как правило, переходит», «как правило, не переходит»), так и глобальные закономерности («чередуется по определённому закону» и т.п.). История поведения инструкций условного перехода записывается в специальных структурах, обычно называемых «таблицами истории переходов» (Branch History Table, BHT). Современные механизмы предсказания переходов обеспечивают правильное предсказание более чем в 90 процентах случаев.

Перечислим некоторые механизмы, используемые в новом процессоре P8, имеющем наиболее совершенную систему предсказания переходов:

* сочетание локального и глобального механизмов для предсказания «обычных» инструкций перехода с учётом истории их поведения;
* статический предсказатель для инструкций, совершающих переход в первый раз, основанный на эмпирических закономерностях (например, «переход назад» обычно предсказывается как совершённый, так как может означать переход по циклу, а «переход вперёд» — как несовершённый);
* предсказатель коротких циклов, распознающий такие переходы и определяющий число итераций цикла (позволяет правильно предсказать момент выхода из цикла);
* предсказатель косвенных переходов, определяющий целевые адреса для различных исполнений инструкции перехода (с учётом возможного чередования этих адресов);

предсказатель целевых адресов для инструкций выхода из подпрограммы, использующий небольшой аппаратный стек для запоминания адресов возврата (Return Address Stack) для эффективной отработки инструкций Call — Return.

В других процессорах компании Intel используется только часть перечисленных механизмов. Эти механизмы совершенствуются с каждым новым поколением процессоров.

В процессорах AMD K8 и IBM PPC970 используются более простые механизмы предсказания обычных переходов, и отсутствуют механизмы предсказания циклов и чередующихся косвенных переходов.

Если после формирования анализируемых признаков оказалось, что направление перехода выбрано верно, все полученные результаты переписываются из буфера по месту назначения, а выполнение программы продолжается в обычном порядке. Если направление перехода предсказано неверно, то буфер результатов очищается. Также очищается и конвейер, содержащий команды, находящиеся на разных этапах обработки, следующие за командой условного перехода. При этом аннулируются результаты всех уже выполненных этапов этих команд. Конвейер начинает загружаться с первой команды другой ветви программы. Так как конвейерная обработка эффективна при большом числе последовательно выполненных команд, то перезагрузка конвейера приводит к значительным потерям производительности. Поэтому вопросам эффективного предсказания направления ветвления разработчики всех процессоров уделяют большое внимание.

Методы предсказания переходов делятся на статические и динамические. При использовании статических методов до выполнения программы для каждой команды условного перехода указывается направление наиболее вероятного ветвления. Это указание делается или программистом с помощью специальных средств, имеющихся в некоторых языках программирования, по опыту выполнения аналогичных программ либо результатам тестового выполнения программы, или программой-компилятором по заложенным в ней алгоритмам. Статические методы предсказания ветвлений слишком упрощены; они предписывают всегда выполнять или не выполнять определенные типы переходов. В некоторых процессорах (не принадлежащих к семейству x86) команды содержат «намек» на направление предполагаемого перехода, который компилятор может сделать на основе ожидаемого им поведения программы.

Но в целом более эффективное решение — динамический алгоритм предсказания ветвлений, который учитывает направления переходов, реализовывавшиеся этой командой при выполнении программы. Например, подсчитывается количество переходов, выполненных ранее по тому или иному направлению, и на основании этого определяется направление перехода при следующем выполнении данной команды. Динамический алгоритм предсказания ветвлений на самом деле оценивает поведение команд перехода за предшествующий период времени (поскольку один и тот же переход часто выполняется более чем один раз, например, в цикле). Благодаря информации о предыстории предсказания будущих ветвлений могут делаться гораздо более точно. Таблица предсказания ветвлений организуется по ассоциативному принципу, подобно кэш-памяти, ее элементы доступны по адресу команды, ветвление которой предсказывается. В некоторых реализациях элемент таблицы предсказания ветвления является счетчиком, значение которого увеличивается при правильном предсказании и уменьшается при неправильном. При этом значение счетчика определяет преобладающее направление ветвлений. Если требуется осуществить смену значения счетчика команд, то необходим, по крайней мере, один такт для распознавания команды ветвления, модификации счет­чика команд и выборки команды по заданному значению счетчика команд. Эти за­держки вызывают пустые такты в конвейерах процессора. Более сложные решения используют буферы, содержащие наборы команд для двух возможных результатов ветвлений.

Возможно также использование «отложенных переходов», когда одна или не­сколько команд после команды ветвления выполняются безусловно.

В момент определения действительного значения условия ветвления вносится изменение в историю ветвления. Если предсказание было неверным, то должна ини­циироваться выборка правильных команд. Результаты команд, которые были услов­но выполнены, должны быть аннулированы.

Механизм предсказания переходов работает одновременно с декодером инструкций и независимо от него. Благодаря эффективной реализации предсказания адреса перехода в процессорах P-III, P-M, P-M2, P8 и K8 при правильном предсказании теряется всего 1 такт. Это означает, что минимальное время, затрачиваемое на итерацию цикла (либо на один переход в цепочке переходов) составляет 2 такта. По существу, предсказатель переходов в таком цикле (или цепочке) работает в своём независимом цикле, состоящем из двух стадий — предсказания и считывания нового блока кэша — а декодер и прочие подсистемы процессора обрабатывают инструкции из вновь считываемых блоков. Поскольку предсказатель переходов «просматривает» целый блок, который может содержать большое число инструкций, то он может «опережать» декодер в своём просмотре. Благодаря этому переход может быть совершён раньше, чем исчерпаются инструкции в текущем блоке, и указанной потери такта не произойдёт — этот такт будет скрыт на фоне бесперебойной работы декодера.

В процессоре PPC970 предсказатель переходов работает менее эффективно — при правильном предсказании теряется 2 такта, а минимальное время итерации цикла составляет 3 такта. Хотя предсказатель просматривает инструкции с некоторым опережением, это может лишь частично скрыть потерю указанных двух тактов, и в результате эффективность исполнения перехода окажется ниже, чем в других процессорах.

Когда инструкция перехода попадёт в функциональное устройство для исполнения, будет выяснено, правильно предсказан этот переход, или нет. В момент её отставки при неправильном предсказании перехода все последующие инструкции будут отменены, и начнётся считывание инструкций из I-кэша по правильному адресу. Такую процедуру называют сбросом конвейера, а время (в тактах), которое было потрачено на выполнение инструкции перехода с момента её считывания из кэша, называют длиной конвейера непредсказанного перехода. Это время характеризует чистую потерю в идеальных условиях, когда инструкция проходила через все этапы «гладко» и нигде не задерживалась по внешним причинам. В реальных условиях потеря на неправильно предсказанный переход может оказаться выше.

Длина конвейера непредсказанного перехода не всегда указывается в документации и известна весьма приблизительно. Её довольно трудно замерить, так как современные предсказатели переходов работают достаточно эффективно и не позволяют добиться гарантированной доли неправильных предсказаний в тестах. Можно дать следующие примерные оценки длины конвейера: P-III — 11, P-M — 12, P-4 — 20, P-4E — 30, P8 — 14, K8 — 11, PPC970 — 13. Нужно учесть, что в процессорах P-4 и P-4E длина такта меньше, чем в других процессорах, и потеря на непредсказанный переход, выраженная в «нормализованных» тактах с учётом соотношения 1:1.4, составит соответственно 15 и 22.

6 Матричные процессоры

Конвейеры и суперскалярная архитектура обычно повышают скорость работы всего лишь в 5-10 раз. Чтобы увеличить производительность в 50, 100 и более раз, нужно создавать компьютеры с несколькими процессорами.

В любой параллельной компьютерной системе процессоры, выполняющие разные части единого задания, должны как-то взаимодействовать друг с другом, чтобы обмениваться информацией. Как именно должен происходить обмен? Для этого было предложено и реализовано две стратегии: мультипроцессоры и мультикомпьютеры. Ключевое различие между стратегиями состоит в наличии или отсутствии общей памяти. Это различие сказывается как на конструкции, устройстве и программировании таких систем, так и на их стоимости и размерах.

6.1 Матричные процессоры

Многие задачи в физических и технических науках предполагают использование массивов или других упорядоченных структур. Часто одни и те же вычисления могут производиться над разными наборами данных в одно и то же время. Упорядоченность и структурированность программ, предназначенных для выполнения такого рода вычислений, очень удобны в плане ускорения вычислений за счет параллельной обработки команд.

Матричный процессор (array processor) состоит из большого числа сходных процессоров, которые выполняют одну и ту же последовательность команд применительно к разным наборам данных. Первым в мире таким процессором был ILLIAC IV (Университет Иллинойс). Схематически он изображен на рисунке 6.1. Первоначально предполагалось сконструировать машину, состоящую из четырех квадрантов, каждый из которых содержал матрицу размером 8 х 8 из блоков процессор/память. Для каждого квадранта имелся один блок контроля. Он рассылал команды, которые выполнялись всеми процессорами одновременно, при этом каждый процессор использовал собственные данные из собственной памяти (загрузка данных происходила при инициализации). Это решение, значительно отличающееся от стандартной фон-неймановской машины, иногда называют архитектурой SIMD (Single Instruction-stream Multiple Data-stream - один поток команд с несколькими потоками данных). Из-за очень высокой стоимости был построен только один такой квадрант, но он мог выполнять 50 млн операций с плавающей точкой в секунду. Если бы при создании машины использовалось четыре квадранта, она могла бы выполнять 1 млрд операций с плавающей точкой в секунду, и вычислительные возможности такой машины в два раза превышали бы возможности компьютеров всего мира.

6.2 Векторный процессор

С точки зрения программиста, векторный процессор (vector processor) очень похож на матричный. Как и матричный, он чрезвычайно эффективен при выполнении последовательности операций над парами элементов данных. Однако в отличие от матричного процессора, все операции сложения выполняются в одном блоке суммирования, который имеет конвейерную структуру. Компания Cray Research, основателем которой был Сеймур Крей, выпустила множество моделей векторных процессоров, начиная с модели Сгау-1 A974.

Оба типа процессоров работают с массивами данных. Оба они выполняют одни и те же команды, которые, например, попарно складывают элементы двух векторов. Однако если у матричного процессора столько же суммирующих устройств, сколько элементов в массиве, векторный процессор содержит векторный регистр, состоящий из набора условных регистров. Эти регистры загружаются из памяти единственной командой, которая фактически делает это последовательно. Команда сложения попарно складывает элементы двух таких векторов, загружая их из двух векторных регистров в суммирующее устройство с конвейерной структурой. В результате из суммирующего устройства выходит другой вектор, который либо помещается в векторный регистр, либо сразу используется в качестве операнда при выполнении другой операции с векторами.

Матричные процессоры в настоящее время не выпускаются, но принцип, на котором они основаны, по-прежнему актуален. Аналогичная идея применяется в наборах ММХ- и SSE-команд процессоров Pentium 4, и она успешно решает задачу ускоренного выполнения мультимедийных программ. В этом отношении компьютер ILLIAC IV можно считать одним из прародителей процессора

Pentium 4.

6.3 Внутрипроцессорная многопоточность

Для всех современных конвейеризованных процессоров характерна одна и та же проблема - если при запросе к памяти слово не обнаруживается в кэшах первого и второго уровней, на загрузку этого слова в кэш уходит длительное время, в течение которого конвейер простаивает. Одна из методик решения этой проблемы называется внутрипроцессорной многопоточностью (on-chip multithreading). Она позволяет процессору одновременно управлять несколькими программными потоками и тем самым маскировать простои. Вкратце принцип можно изложить так: если программный поток 1 блокируется, процессор может обеспечить полную загрузку аппаратуры, запустив программный поток 2.

Основополагающая идея проста, реализуется она разными способами. Первый из них, называемый мелкомодульной многопоточностью (fine-grained multithreading), применительно к процессору, способному вызывать одну команду за такт, иллюстрирует рисунок 6.2. На рисунке 6.2 а-в изображено три программных потока (А, В, С), соответствующих 12 машинным циклам. В ходе первого цикла поток А выполняет команду A1. Поскольку эта команда завершается за один цикл, при наступлении второго цикла запускается команда А2. Ее обращение в кэш первого уровня оказывается неудачным, поэтому до извлечения нужного слова из кэша второго уровня проходит два цикла. Исполнение потока продолжается в цикле 5. Как показано на рисунке, потоки В и С также регулярно простаивают. В рамках такого решения вызов последующей команды до завершения предыдущей не осуществляется. Точнее, при наличии сложного счетчика обращений в некоторых случаях это допустимо, но такую возможность мы для простоты исключаем.

При мелкомодульной многопоточности простой маскируется путем исполнения потоков "по кругу", то есть в смежных циклах запускаются разные потоки (рисунок 6.2 г). К моменту наступления цикла 4 обращение к памяти, инициированное командой A1, завершается, поэтому даже если команде A2 нужен результат команды A1, она запускается. В таком случае максимальная продолжительность простоя составляет два цикла, то есть при наличии трех программных потоков простаивающая операция все равно завершается вовремя. При простое в 4 цикла для беспрерывной работы понадобилось бы 4 программных потока, и т. д.

Поскольку разные программные потоки никак друг с другом не связаны, каждому из них нужен свой набор регистров. Он должен быть указан для каждой вызываемой команды, и тогда аппаратное обеспечение будет знать, к какому набору регистров при необходимости нужно обращаться. Следовательно, максимальное число одновременно исполняемых программных потоков определяется в период разработки микросхемы.

Обращениями к памяти причины простоя не ограничиваются. Иногда для исполнения следующей команды требуется результат предыдущей команды, который еще не вычислен. В других случаях команда вызвана быть не может, так как она следует за условным переходом, направление которого еще неизвестно. Общее правило формулируется так: если в конвейере k ступеней, но по кругу можно запустить, по меньшей мере, k программных потоков, то в одном потоке в любой отдельно взятый момент не может выполняться более одной команды, поэтому конфликты между ними исключены. В такой ситуации процессор может работать на полной скорости, без простоя.

Естественно, далеко не всегда число доступных потоков равно числу ступеней конвейера, поэтому некоторые разработчики предпочитают методику, называемую крупномодульной многопоточностью (coarse-grained multithreading), которую иллюстрирует рисунок 6.2, д. В данном случае программный поток А продолжает выполняться последовательно, вплоть до простоя. При этом теряется один цикл. Далее происходит переключение на первую команду программного потока B (B1). Так как эта команда сразу переходит в состояние простоя, в цикле 6 выполняется уже команда C1. Так как каждый раз при простое команды теряется один цикл, по своей эффективности крупномодульная многопоточность, казалось бы, уступает мелкомодульной, однако у нее есть одно существенное преимущество - за счет меньшего числа программных потоков значительно сокращается расход ресурсов процессора. При недостаточном количестве активных потоков эта методика оптимальна.

Судя по нашему описанию, при крупномодульной многопоточности просто выполняется переключение между потоками, однако это - не единственный предусматриваемый данной методикой вариант действий. Есть возможность немедленного переключения с команд, которые потенциально способны вызвать простой (например, загрузка, сохранение и переходы), без выяснения, действительно ли намечается простой. Эта стратегия позволяет переключаться раньше обычного (сразу после декодирования команды) и исключает бесконечные циклы. Иными словами, исполнение продолжается до того момента, пока не обнаружится возможность возникновения проблемы, после чего следует переключение. Такие частые переключения роднят крупномодульную многопоточность с мелкомодульной.

Вне зависимости от используемого варианта многопоточности, необходимо как-то отслеживать принадлежность каждой операции к тому или иному программному потоку. В рамках мелкомодульной многопоточности каждой операции присваивается идентификатор потока, поэтому при перемещениях по конвейеру ее принадлежность не вызывает сомнений. Крупномодульная многопоточность предусматривает возможность очистки конвейера перед запуском каждого последующего потока. В таком случае четко определяется идентичность потока, исполняемого в данный момент. Естественно, данная методика эффективна только в том случае, если паузы между переключениями значительно больше времени, необходимого для очистки конвейера.

Все сказанное относится к процессорам, способным вызывать не более одной команды за тактовый цикл. Однако мы знаем, что для современных процессоров это ограничение не актуально. Применительно к изображению на рисунке 6.3 мы допускаем, что процессор может вызывать по 2 команды за цикл, однако утверждение о невозможности запуска последующих команд в случае простоя предыдущей остается в силе. Рисунок 6.3, а иллюстрирует механизм мелкомодульной многопоточности в сдвоенном суперскалярном процессоре. Как видно, в потоке А первые две команды запускаются во время первого цикла, однако в потоке В во втором цикле запускается только одна команда.

В суперскалярных процессорах есть еще один способ организации многопоточности - так называемая синхронная многопоточность (simultaneous multithreading), которую иллюстрирует рисунок 6.3, в. Эта методика представляет собой усовершенствованный вариант крупномодульной многопоточности, где каждый программный поток может запускать по две команды за такт, однако в случае простоя с целью обеспечения полной загрузки процессора запускаются команды следующего потока. При синхронной многопоточности полностью загружаются все функциональные блоки. В случае невозможности запуска команды из-за занятости функционального блока выбирается команда из другого потока. На рисунке предполагается, что в цикле 11 простаивает команда B8, поэтому в цикле 12 запускается команда С7.

6.4 Многопоточность в Pentium 4

Разобравшись с теорией многопоточности, рассмотрим практический пример - Pentium 4. Уже на этапе разработки этого процессора инженеры Intel продолжали работу над повышением его быстродействия без внесения изменений в программный интерфейс. Рассматривалось пять простейших способов:

* повышение тактовой частоты;
* размещение на одной микросхеме двух процессоров;
* введение новых функциональных блоков;
* удлинение конвейера;
* использование многопоточности.

Самый очевидный способ повышения быстродействия заключается в том, чтобы повысить тактовую частоту, не меняя другие параметры. Как правило, каждая последующая модель процессора имеет несколько более высокую тактовую частоту, чем предыдущая. К сожалению, при прямолинейном повышении тактовой частоты разработчики сталкиваются с двумя проблемами: увеличением энергопотребления (что актуально для портативных компьютеров и других вычислительных устройств, работающих на аккумуляторах) и перегревом (что требует создания более эффективных теплоотводов).

Второй способ - размещение на микросхеме двух процессоров - сравнительно прост, но он сопряжен с удвоением площади, занимаемой микросхемой. Если каждый процессор снабжается собственной кэш-памятью, количество микросхем на пластине уменьшается вдвое, но это также означает удвоение затрат на производство. Если для обоих процессоров предусматривается общая кэш-память, значительного увеличения занимаемой площади удается избежать, однако в этом случае возникает другая проблема - объем кэш-памяти в пересчете на каждый процессор уменьшается вдвое, а это неизбежно сказывается на производительности. Кроме того, если профессиональные серверные приложения способны полностью задействовать ресурсы нескольких процессоров, то в обычных настольных программах внутренний параллелизм развит в значительно меньшей степени.

Введение новых функциональных блоков также не представляет сложности, но здесь важно соблюсти баланс. Какой смысл в десятке блоков АЛУ, если микросхема не может выдавать команды на конвейер с такой скоростью, которая позволяет загрузить все эти блоки?

Конвейер с увеличенным числом ступеней, способный разделять задачи на более мелкие сегменты и обрабатывать их за короткие периоды времени, с одной стороны, повышает производительность, с другой, усиливает негативные последствия неверного прогнозирования переходов, кэш-промахов, прерываний и других событий, нарушающих нормальный ход обработки команд в процессоре. Кроме того, чтобы полностью реализовать возможности расширенного конвейера, необходимо повысить тактовую частоту, а это, как мы знаем, приводит к повышенным энергопотреблению и теплоотдаче.

Наконец, можно реализовать многопоточность. Преимущество этой технологии состоит во введении дополнительного программного потока, позволяющего ввести в действие те аппаратные ресурсы, которые в противном случае простаивали бы. По результатам экспериментальных исследований разработчики Intel выяснили, что увеличение площади микросхемы на 5 % при реализации многопоточности для многих приложений дает прирост производительности на 25 %. Первым процессором Intel с поддержкой многопоточности стал Хеоn 2002 года. Впоследствии, начиная с частоты 3,06 ГГц, многопоточность была внедрена в линейку Pentium 4. Intel называет реализацию многопоточности в Pentium 4 гиперпоточностью (hyperthreading).

7 Закон Амдала. Закон Густафсона

7.1 Ускорение, эффективность, загрузка и качество

Параллелизм достигается за счет того, что в составе вычислительной системы отдельные устройства присутствуют в нескольких копиях. Так, в состав процессора может входить несколько АЛУ, и высокая произ­водительность обеспечивается за счет одновременной работы всех этих АЛУ. Второй подход был описан ранее.

Рассмотрим параллельное выполнение программы со следующими характеристиками:

О(n) — общее число операций (команд), выполненных на n-процессорной системе;

Т(n) — время выполнения О(n) операций на n-процессорной системе в виде числа квантов времени.

В общем случае Т(n) < О(n), если в единицу времени n процессорами выполня­ется более чем одна команда, где n > 2. Примем, что в однопроцессорной системе T(1)= О(1).

Ускорение (speedup), или точнее, среднее ускорение за счет параллельного выполнения программы — это отношение времени, требуемого для выполнения наилучшего из последовательных алгоритмов на одном процессоре, и времени параллельного вычисления на n процессорах. Без учета коммуникационных издержек ускорение S(n) определяется как

Как правило, ускорение удовлетворяет условию S(n) n. Эффективность (efficiency) n-процессорной системы — это ускорение на один процессор, определяемое выражением

Эффективность обычно отвечает условию 1/n Е(n) n. Для более реалис­тичного описания производительности параллельных вычислений необходимо промоделировать ситуацию, когда параллельный алгоритм может потребовать больше операций, чем его последовательный аналог.

Довольно часто организация вычислений на n процессорах связана с существенными издержками. Поэтому имеет смысл ввести понятие избыточности (redun­dancy) в виде

Это отношение отражает степень соответствия между программным и аппаратным параллелизмом. Очевидно, что 1 R(n) n.

Определим еще одно понятие, коэффициент полезного использования или утилизации (utilization), как

Тогда можно утверждать, что

Рассмотрим пример. Пусть наилучший из известных последовательных алгоритмов занимает 8 с, а параллельный алгоритм занимает на пяти процессорах 2 с. Тогда:

S(n) = 8/2 = 4;

Е(n) = 4/5 = 0,8;

R(n) = 1/0,8 - 1 = 0,25.

Собственное ускорение определяется путем реализации параллельного алгоритма на одном процессоре.

Если ускорение, достигнутое на n процессорах, равно n, то говорят, что алгоритм показывает линейное ускорение.

В исключительных ситуациях ускорение S(n) может быть больше, чем n. В этих случаях иногда применяют термин суперлинейное ускорение. Данное явление имеет шансы на возникновение в двух следующих случаях:

Последовательная программа может выполняться в очень маленькой памяти, вызывая свопинги (подкачки), или, что менее очевидно, может приводить к большему числу кэш-промахов, чем параллельная версия, где обычно каждая параллельная часть кода выполняется с использованием много меньшего наб­ра данных.

Другая причина повышенного ускорения иллюстрируется примером. Пусть нам нужно выполнить логическую операцию A1 v A2, где как A1 так и А2 имеют значение «Истина» с вероятностью 50%, причем среднее время вычисления Ai, обозначенное как T(Ai), существенно различается в зависимости от того, является ли результат истинным или ложным.

Пусть

Теперь получаем четыре равновероятных случая (Т — «истина», F — «ложь»):

Таким образом, параллельные вычисления на двух процессорах ведут к среднему ускорению:

Отметим, что суперлинейное ускорение вызвано жесткостью последователь­ной обработки, так как после вычисления еще нужно проверить А2. К факторам, ограничивающим ускорение, следует отнести:

* Программные издержки. Даже если последовательные и параллельные алгоритмы выполняют одни и те же вычисления, параллельным алгоритмам присущи добавочные программные издержки — дополнительные индексные вычисления, неизбежно возникающие из-за декомпозиции данных и распределения их по процессорам; различные виды учетных операций, требуемые в параллельных алгоритмах, но отсутствующие в алгоритмах последовательных.
* Издержки из-за дисбаланса загрузки процессоров. Между точками синхронизации каждый из процессоров должен быть загружен одинаковым объемом работы, иначе часть процессоров будет ожидать, пока остальные завершат свои операции. Эта ситуация известна как дисбаланс загрузки. Таким образом, ускорение ограничивается наиболее медленным из процессоров.
* Коммуникационные издержки. Если принять, что обмен информацией и вычисления могут перекрываться, то любые коммуникации между процессорами снижают ускорение. В плане коммуникационных затрат важен уровень гранулярности, определяющий объем вычислительной работы, выполняемой между коммуникационными фазами алгоритма. Для уменьшения коммуникационных издержек выгоднее, чтобы вычислительные гранулы были достаточно крупными, и доля коммуникаций была меньше.

Еще одним показателем параллельных вычислений служит качество параллель­ного выполнения программ — характеристика, объединяющая ускорение, эффективность и избыточность. Качество определяется следующим образом:

Поскольку как эффективность, так и величина, обратная избыточности, представляют собой дроби, то Q(n) S(n). Поскольку Е(n) — это всегда дробь, a R(n) -число между 1 и n, качество Q(n) при любых условиях ограничено сверху величиной ускорения S(n) [4].

7.2 Закон Амдала

В идеальном случае система из n процессоров могла бы ускорить вычисления в n раз. В реальности достичь такого показателя по ряду причин не удается. Главная из этих причин заключается в невозможности полного распараллеливания ни одной из задач. Как правило, в каждой программе имеется фрагмент кода, который принципиально должен выполняться последовательно и только одним из процессоров. Это может быть часть программы, отвечающая за запуск задачи и распределение распараллеленного кода по процессорам, либо фрагмент программы, обеспечивающий операции ввода/вывода. Можно привести и другие примеры, но главное состоит в том, что о полном распараллеливании задачи гово­рить не приходится. Известные проблемы возникают и с той частью задачи, кото­рая поддается распараллеливанию. Здесь идеальным был бы вариант, когда параллельные ветви программы постоянно загружали бы все процессоры системы, причем так, чтобы нагрузка на каждый процессор была одинакова. К сожалению, оба этих условия на практике трудно реализуемы. Таким образом, ориентируясь на параллельную ВС, необходимо четко сознавать, что добиться прямо пропорционального числу процессоров увеличения производительности не удастся, и, естественно, встает вопрос о том, на какое реальное ускорение можно рассчитывать. Ответ на этот вопрос в какой-то мере дает закон Амдала.

Джин Амдал (Gene Amdahl) — один из разработчиков всемирно известной системы IBM 360, в своей работе, опубликованной в 1967 году, предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на многопроцессорной ВС, от числа процессоров и соотношения между последовательной и параллельной частями программы. Показателем сокращения времени вычислений служит такая метрика, как «ускорение». Напомним, что ускорение S— это отношение времени Ts, затрачиваемого на проведение вычислений на однопроцессорной ВС (в варианте наилучшего последовательного алгоритма), ко времени Тр решения той же задачи на параллельной системе (при использовании наилучшего параллельного алгоритма):

Оговорки относительно алгоритмов решения задачи сделаны, чтобы подчеркнуть тот факт, что для последовательного и параллельного решения лучшими могут оказаться разные реализации, а при оценке ускорения необходимо исходить именно из наилучших алгоритмов.

Проблема рассматривалась Амдалом в следующей постановке (рисунок 7.1). Прежде всего, объем решаемой задачи с изменением числа процессоров, участвующих в ее решении, остается неизменным. Программный код решаемой задачи состоит из двух частей: последовательной и распараллеливаемой. Обозначим долю операций, которые должны выполняться последовательно одним из процессоров, через f, где 0f1 (здесь доля понимается не по числу строк кода, а по числу реально выполняемых операций). Отсюда доля, приходящаяся на распараллеливаемую часть программы, составит 1-f. Крайние случаи в значениях f соответствуют полностью параллельным (f=0) и полностью последовательным (f=1) программам. Распараллеливаемая часть программы равномерно распределяется по всем процессорам.

С учетом приведенной формулировки имеем:

В результате получаем формулу Амдала, выражающую ускорение, которое мо­жет быть достигнуто на системе из n процессоров:

Формула выражает простую и обладающую большой общностью зависимость. Характер зависимости ускорения от числа процессоров и доли последовательной части программы показан на рисунке 7.2.

Если устремить число процессоров к бесконечности, то в пределе получаем:

Это означает, что если в программе 10% последовательных операций (то есть f=0,1), то, сколько бы процессоров ни использовалось, убыстрения работы программы более чем в десять раз никак ни получить, да и то, 10 — это теоретическая верхняя оценка самого лучшего случая, когда никаких других отрицательных факторов нет. Следует отметить, что распараллеливание ведет к определенным издержкам, которых нет при последовательном выполнении программы. В качестве примера таких издержек можно упомянуть дополнительные операции, связанные с распределением программ по процессорам, обмен информацией между процессорами [4].

7.3 Закон Густафсона

Решая на вычислительной системе из 1024 процессоров три больших задачи, для которых доля последовательного кода f лежала в пределах от 0,4 до 0,8%, Джон Густафсон из NASA Ames Research получил значения ускорения по сравнению с однопроцессорным вариантом, равные соответственно 1021,1020 и 1016. Согласно закону Амдала для данного числа процессоров и диапазона f, ускорение не должно было превысить величины порядка 201. Пытаясь объяснить это явление, Густафсон пришел к выводу, что причина кроется в исходной предпосылке, лежащей в основе закона Амдала: увеличение числа процессоров не сопровождается увеличением объема решаемой задачи. Реальное же поведение пользователей существенно отличается от такого представления. Обычно, получая в свое распоряжение более мощную систему, пользователь не стремится сократить время вычислений, а, сохраняя его практически неизменным, старается пропорционально мощности ВС увеличить объем решаемой задачи. И тут оказывается, что наращивание общего объема программы касается главным образом распараллеливаемой части программы. Это ведет к сокращению значения f. Примером может служить решение дифференциального уравнения в частных производных. Если доля последовательного кода составляет 10% для 1000 узловых точек, то для 100 000 точек доля последовательного кода снизится до 0,1%. Сказанное иллюстрирует рисунок 7.3, который отражает тот факт, что, оставаясь практически неизменной, последовательная часть в общем объеме увеличенной программы имеет уже меньший удельный вес.

Было отмечено, что в первом приближении объем работы, которая может быть произведена параллельно, возрастает линейно с ростом числа процессоров в системе. Для того чтобы оценить возможность ускорения вычислений, когда объем последних увеличивается с ростом количества процессоров в системе (при посто­янстве общего времени вычислений), Густафсон рекомендует использовать выра­жение, предложенное Е. Барсисом (Е. Barsis):

Данное выражение известно как закон масштабируемого ускорения или закон Густафсона (иногда его называют также законом Густафсона-Барсиса). В заключение отметим, что закон Густафсона не противоречит закону Амдала. Различие состоит лишь в форме утилизации дополнительной мощности ВС, возникающей при увеличении числа процессоров [4].

вывод

Развитие вычислительной техники характеризуется тем, что на каждом этапе новых разработок требования к производительности значительно превышают возможности элементной базы.

Это обусловлено задачами сложных систем управления в реальном времени, централизованным решением задач в сетях, имитационным моделированием сложных процессов (например, в ядерной физике), оперативным планированием и управлением. Такие задачи требуют концентрации вычислительных мощностей, постоянно поддерживая высокую актуальность проблемы создания супер-ЭВМ.

Добиваться повышения производительности компьютеров только за счет увеличения тактовой частоты становится все сложнее, так как появляется проблема отвода тепла. Поэтому разработчики обратили свое внимание на параллелизм как на средство ускорения вычислений. Параллелизм может вводиться на разных уровнях, как на самых нижних, где элементы очень жестко связаны друг с другом, так и на верхних, где связи весьма слабые.

Параллелизм на уровне команд имеет место, когда обработка нескольких команд или выполнение различных этапов одной и той же команды может перекрываться во времени. Разработчики вычислительной техники прибегают к методам, известным под общим названием «совмещения операций», при котором аппарату­ра ВМ в любой момент времени выполняет одновременно более одной операции.

В работе мы рассмотрели принципы организации параллелизма на уровне команд, ознакомились с работой конвейера, суперскалярного конвейера, матричного процессора, убедились в том, что параллелизм – эффективный способ повышения производительности при неизменной тактовой частоте процессора.

список литературы

1. Корнеев В.В. Вычислительные системы.-М.:Гелиос APB, 2004.-512с., ил.- с. 34-46
2. Таненбаум, Э. Архитектура компьютера/Пер. с англ.- 4-е изд. Учебник для вузов.-СПб.: Питер, 2003 698 с. : ил.
3. Барский А.Б. Параллельные информационные технологии: Учебное пособие/А.Б. Барский.-М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2007.-503 с.: ил.,таб.-(серия «Основы информационных технологий»)- с.20-28, с.56-58.
4. Цилькер, Б. Я. Организация ЭВМ и систем : Учебник для вузов.-СПб.: Питер, 2007.-668с.: ил- с. 481-492
5. Куприянов М.С., Петров Г.А., Пузанков Д.В. Процессор Pentium: Архитектура и программирование. /ГЭТУ-СПб., 1995.-277с -с. 11-17
6. Гук М, Юров В. Процессоры Pentium 4, Atlon и Duron.-СПб.: Питер, 2001.-512 с.: ил- с. 42-46
7. Головкин Б.А. Параллельные вычислительные системы. - М.: Наука, 1980. - 518 с.
8. Шпаковский Г.И. Архитектура параллельных ЭВМ: Учеб. пособие для вузов. - Мн.: Университетское, 1989. - 192 с.
9. Коуги П.М. Архитектура конвейерных ЭВМ / Пер. с англ. - М.: Радио и связь, 1985. - 360 с.
10. Хокни З., Джессхоуп К. Параллельные ЭВМ. - М.: Радио и связь, 1986. - 389 с.
11. http://www.tver.mesi.ru/e-lib/res/628/11/2.html - Интернет-университет информационных технологий - дистанционное образование