Конвейерная организация работы микропроцессора

Выполнение каждой команды складывается из ряда последовательных этапов, суть которых не меняется от команды к команде. С целью увеличения быстродействия процессора и максимального использования всех его возможностей в современных микропроцессорах используется **конвейерный принцип обработки информации**. Этот принцип подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. По очередному тактовому импульсу каждая команда в конвейере продвигается на следующую стадию обработки, выполненная команда покидает конвейер, а новая поступает в него.

В различных процессорах количество и суть этапов различаются.

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

1. IF (INsTRuction Fetch) - считывание команды в процессор;
2. ID (INsTRuction DecodINg) - декодирование команды;
3. OR (Operand ReadINg) - считывание операндов;
4. EX (ExecutINg) - выполнение команды;
5. WB (Write Back) - запись результата.

Выполнение команд в таком конвейере представлено в табл. 9.1.

Таблица 9.1. Порядок выполнения команд в 5-ступенчатом конвейре.

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

Оценка производительности идеального конвейера

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

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 t = 5 единиц, получим время такта:

$$T = max\{ T_{IF} = 20, T_{ID} = 15, T_{OR} = 20, T_{EX} = 25, T_{WB} = 20 \} + \Delta t = 30$$

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

При последовательной обработке время выполнения N команд составит:

$$T_{\text{посл}} = N \times (T_{IF} + T_{ID} + T_{OR} + T_{EX} + T_{WB}) = 100N$$

Анализ табл. 9.1 п показывает, что при конвейерной обработке после того, как получен результат выполнения первой команды, результат очередной команды появляется в следующем такте работы процессора. Следовательно:

$$T_{\text{конв}} = 5T + (N - 1) \times T$$

Примеры длительности выполнения некоторого количества команд при последовательной и конвейерной обработке приведены в табл. 9.2.

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 за рассматриваемый период команд. Сокращение длительности такта может достигаться разбиением выполнения команды на большое число этапов, каждый из которых включает в себя относительно простые операции и поэтому будет выполняться за более короткий промежуток времени. Так, если в микропроцессоре Pentium длина конвейера составляла 5 ступеней (при максимальной тактовой частоте 200 МГц), то в процессорах Pentium 4 на ядре Northwood длина конвейера составляла 20 ступеней, а на ядре Prescott она увеличена до 31 ступени при максимальной тактовой частоте 3,8 ГГц.

Таблица 9.2. Оценка эффективности конвейерной обработки

Количество команд	**Время**	
	при последовательном выполнении	**при конвейерном выполнении**
1	100	150
2	200	180
10	1000	420
100	10000	3120

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

Конфликты в конвейере и способы минимизации их влияния на производительность процессора

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 табл. 9.3.

Таблица 9.3. Конвейерная обработка при задержке команды i+1 на этапе EX

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

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

Если какой-то блок конвейера вносит задержку, то тормозится работа всего конвейера. Образуемый при этом "пузырь" должен пройти от места своего возникновения до самого конца конвейера (если, например, возникла задержка на ступени считывания команды, то в следующем такте блок декодирования от него ничего не получит, а через 3 такта соответственно блок сохранения результатов ничего не получит от блока выполнения). Таким образом, скорость конвейера определяется скоростью самой медленной его ступени.

Этой ситуации можно было бы избежать 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 действия, приводящего к появлению "пузырей" (например, использовать матричные схемы умножения). Но это приведет к усложнению схемы процессора и сокращению на кристалле места для реализации на этой БИС других, функционально более важных узлов. Так как представленная в табл. 9.3 сситуация 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 таких конфликтов служит конфликт из-за доступа к запоминающим устройствам. Из табл. 9.1 ввидно, 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. В запоминающих устройствах в современных микропроцессорах с этой целью 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 нескольким каналам для записи и считывания информации. Например, в микропроцессоре Itanium к блоку регистров общего назначения допускается одновременное обращение на выполнение 8 операций чтения и 6 операций записи. Конфликты из-за исполнительных устройств обычно сглаживаются введением в состав микропроцессора дополнительных блоков. Так, в микропроцессоре Pentium 4 для обработки целочисленных данных предусмотрено 4 АЛУ.

При этом появляется возможность параллельной обработки информации в нескольких конвейерах. Процессоры, имеющие в своем составе более одного конвейера, называются **суперскалярными**.

Недостатком суперскалярных микропроцессоров является необходимость синхронного продвижения команд в каждом из конвейеров. В табл. 9.4 п представлена последовательность выполнения команд в микропроцессоре, имеющем два конвейера, при условии, что команде К1 требуется 3 такта на этапе EX.

Таблица 9.4. Обработка команд в микропроцессоре с двумя конвейерами

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 порядке, отличающемся от того, который предусмотрен программой: К2-К4-К1-К6-...

Следовательно, для обеспечения правильной работы суперскалярного микропроцессора при возникновении затора в одном из конвейеров должны приостанавливать свою работу и другие. В противном случае может нарушиться исходный порядок завершения команд программы. 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, а аппаратные средства микропроцессора должны гарантировать, что результаты выполненных команд будут записаны в приемник в том порядке, в котором команды записаны в программе. Это обеспечивается путем использования так называемого **принципа неупорядоченного выполнения команд**. Суть его заключается в следующем. Блок выборки и декодирования выбирает команды из памяти и заносит их в буфер команд. По мере готовности операндов и исполнительного блока соответствующего типа команды извлекаются из буфера для обработки. Порядок их исполнения может отличаться от предписанного программой.

Результаты этапа выполнения команды 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. Преимущества такого подхода очевидны: команды максимально используют возможности всех конвейеров, присутствующих в микроархитектуре микропроцессора, что обеспечивает его максимальную производительность.

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 табл. 9.1, 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 больше.

Простейший способ разрешения этой ситуации - использование так называемого метода выжидания. Он заключается в замораживании операций в конвейере путем блокировки выполнения любой команды, следующей за командой условного перехода, до тех пор, пока не станет известным направление перехода. Привлекательность такого решения заключается в его простоте. Главный недостаток - резкое уменьшение

преимуществ конвейерной обработки. 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.

Можно несколько улучшить эту ситуацию, используя схему "задержанных переходов". При этом на стадии компиляции компилятор таким образом структурирует получаемый объектный код, чтобы сделать команды, следующие за командой перехода, действительными и полезными (рис. 9.1).

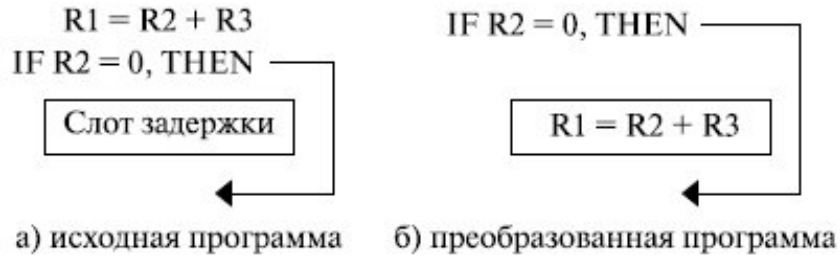

Рис. 9.1. Организация задержанного перехода

Слот задержки заполняется независимой командой, находящейся перед командой условного перехода. Эта команда выполняется за то время, пока микропроцессор формирует истинное условие, по которому должно быть определено направление выполнения программы. Одной из основных трудностей в этом подходе является определение точного времени выполнения команды, вносимой в слот задержки. Аппаратура должна гарантировать реальное выполнение этих команд перед выполнением собственно перехода.

Более эффективными для снижения потерь от конфликтов по управлению являются методы **предсказания переходов**. Они призваны максимально ускорить определение адреса команды, выполняемой после команды перехода.

Так как преимущества конвейерной обработки проявляются при большом числе последовательно выполненных команд, перезагрузка конвейера приводит к значительным потерям производительности. Поэтому вопросам эффективного предсказания направления ветвления разработчики всех микропроцессоров уделяют большое внимание.

Среди основных достоинств практически каждого нового микропроцессора производители анонсируют "улучшенный блок предсказания переходов". Суть конкретных механизмов, обеспечивающих эти улучшения, как правило, не детализируется. Однако здесь все-таки можно выделить несколько основных подходов.

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 программойкомпилятором по заложенным в ней алгоритмам. Подобный подход реализован, например, в HP PA-8x00. Также это может делать и сам программист по опыту выполнения аналогичных программ либо по результатам тестового выполнения программы. Например, в системе команд микропроцессора Itanium для этого предназначена специальная команда.

Суть данного метода заключается в том, что при выполнении команды условного перехода специальный блок микропроцессора 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, 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.

Если после формирования анализируемых признаков оказалось, что направление перехода выбрано верно, все полученные результаты переписываются из буфера по месту назначения и выполнение программы

продолжается в обычном порядке. Если направление перехода предсказано неверно, все инструкции, выбранные после перехода, помечаются, согласно интеловской терминологии, как **поддельные** (bogus INsTRuctions).

При этом буфер результатов и конвейер, содержащий команды, которые следуют за командой условного перехода и находятся на разных этапах обработки, - очищаются. Аннулируются результаты всех уже выполненных этапов этих команд. Конвейер начинает загружаться с первой команды другой ветви программы.

Следует отметить, что конфликты по управлению не исчерпываются только проблемами, связанными с командами условных переходов. Они возникают при выполнении всех команд, меняющих значение счетчика команд. Это хорошо видно из табл. 9.1. Если команда i является командой такого типа (например, команда безусловного перехода), то адрес перехода будет вычислен ею в такте 5, в то время как уже в такте 2 необходимо выбирать в микропроцессор следующую команду по этому адресу.

Методы динамического предсказания реализуются при выполнении программы в микропроцессоре. Они осуществляют предсказание направления переходов на основании результатов предыдущих выполнений данной команды.

При использовании этих методов для команд условных переходов анализируется предыстория переходов - результаты нескольких предыдущих команд ветвления по данному адресу. В этом случае возможно определение чаще всего реализуемого направления ветвления, а также выявление чередующихся переходов.

Для команд безусловных переходов однажды вычисленный целевой адрес сохраняется в специальной памяти BTB (Branch Target Buffer), откуда он извлекается сразу же при декодировании данной команды.

Аналогичный подход используется для команд вызова - возврата из процедуры (анализ связок CALL - RETURN).

Проблемы реализации точного прерывания в конвейере

Обработка прерываний при конвейерной организации работы МП оказывается достаточно сложной из-за того, что совмещенное выполнение команд затрудняет определение возможности безопасного изменения состояния машины произвольной командой. В конвейерной системе команда выполняется по этапам. В ходе выполнения отдельных этапов команда может изменить состояние процессора. Тем временем возникшее прерывание может вынудить машину прервать обработку еще не завершенных команд.

При переходе на программу - обработчик прерывания необходимо надежно очистить конвейер и сохранить состояние процессора таким, чтобы повторное выполнение команды после возврата из прерывания осуществлялось корректно.

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.

Рис. 9.2. Конфликт по данным типа RAW

Существует несколько типов конфликтов по данным:

1. Конфликты типа RAW (Read After Write - чтение после записи):

   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 на рис. 9.2.

   Пусть выполняемые команды имеют следующий вид:

```
i) ADD R1,R0; R1=R1+R0
i+1=j) SUB R2,R1; R2=R2-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.

   Конфликты типа **RAW** обусловлены именно конвейерной организацией обработки команд. Они называются истинными взаимозависимостями.

   Уменьшение влияния конфликта типа **RAW** обеспечивается методом, который называется пересылкой или продвижением данных (data forwardINg), обходом (data bypassINg), иногда закороткой (short-circuitINg).

   В этом случае результаты, полученные на выходах исполнительных устройств, помимо входов приемника результата передаются также на входы всех исполнительных устройств микропроцессора (рис. 9.3).

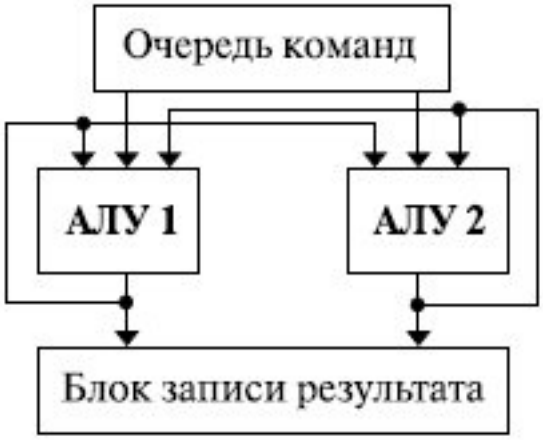

Рис. 9.3. Уменьшение влияния конфликта типа RAW методом продвижения данных

Если устройство управления обнаруживает, что полученный какойлибо командой 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.

Главной причиной двух других типов конфликтов по данным является возможность неупорядоченного выполнения команд в современных микропроцессорах, то есть выполнения команд не в том порядке, в котором они записаны в программе (ложные взаимозависимости).

2. Конфликты типа WAR (Write After Read - запись после чтения):

   команда j пытается записать результат в приемник, прежде чем он считается оттуда командой i. При этом команда i может получить некорректное новое значение операнда:

```
i) ADD R1,R0; R1=R1+R0
i+1 = j) SUB R0,R2; R0=R0-R2
```

Этот конфликт возникнет в случае, если команда j вследствие неупорядоченного выполнения завершится раньше, чем команда i прочитает старое содержимое регистра R0.

3. Конфликты типа **WAW** (Write After Write - запись после записи):

   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:

```
i) ADD R1,R0; R1=R1+R0
. . .
j) SUB R1,R2; R1=R1-R2
```

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

Часть конфликтов по данным может быть снята специальной методикой планирования компилятора. В простейшем случае компилятор просто планирует распределение команд в базовом блоке. Базовый блок представляет собой линейный участок последовательности программного кода с одним входом и одним выходом, в котором отсутствуют внутренние команды перехода. Поскольку в таком блоке каждая команда будет выполняться, если выполняется первая из них, - можно построить граф зависимостей этих команд и упорядочить их так, чтобы минимизировать приостановки конвейера. Эта техника называется планированием загрузки конвейера (pIPelINe schedulINg) или планированием потока команд (INsTRuction schedulINg).

Например, для оператора A = B + C компилятор, скорее всего, сгенерирует следующую последовательность команд:

1. Rb = B.
2. >Rc = C.
3. Ra = Rb + Rc.
4. A = Ra.

Очевидно, выполнение команды (3) должно быть приостановлено до тех пор, пока не станет доступным поступающий из памяти операнд C.

Для данного простейшего примера компилятор никак не может улучшить ситуацию, однако в ряде более общих случаев он может реорганизовать последовательность команд так, чтобы избежать приостановок конвейера.

Пусть, например, имеется последовательность операторов:

```
A = B + C;
D = E - F.
```

В этом случае компилятор может сгенерировать следующую последовательность команд, выполнение которой не приведет к приостановке конвейера:

1. Rb = B.
2. Rc = C.
3. Re = E.
4. Ra = Rb + Rc.
5. Rf = F.
6. A = Ra.
7. Rd = Re - Rf.
8. D = Rd.

Заметим, что использование разных регистров для первого и второго компилируемого оператора было достаточно важным для реализации такого правильного планирования. В частности, если переменная e была бы загружена в тот же самый регистр, что b или c, такое планирование не было бы корректным. В общем случае планирование конвейера может потребовать увеличенного количества регистров.

Для простых конвейеров стратегия планирования на основе базовых блоков вполне удовлетворительна, однако когда конвейеризация становится более интенсивной и действительные задержки конвейера растут, требуются более сложные алгоритмы планирования.

Зачастую зависимость по данным не является необходимой - просто так уж повелось у программистов: чем меньше переменных (и регистров в программах на ассемблере) использует программа - тем лучше. В результате зачастую получается, что вся программа использует один-два регистра с зависимостью по данным чуть ли не в каждой паре команд.

Устранение конфликтов, связанных с ложными взаимозависимостями данных, часто возможно путем **переименования регистров** (register renamINg). Суть этого механизма заключается в следующем. Процессоры, использующие переименование регистров, фактически имеют больше восьми регистров, определяемых архитектурой x86 или IA32. При этом если какой-либо команде требуется использовать регистр, процессор динамически ставит в соответствие этому логическому (архитектурному) регистру один из более многочисленных физических регистров. Если другая команда пытается обратиться к тому же логическому регистру, процессор для предотвращения конфликта может поставить ему в соответствие другой физический регистр. Такие переименования действуют, пока команды продвигаются по конвейерам.

Таким образом, каждый раз, когда команда прямо или косвенно пишет в регистр, ей выделяется новый физический регистр. В МП имеется таблица отображения логических (видимых программисту) регистров на физические (видимые только процессору). Когда команде выделяется новый физический регистр, таблица обновляется: логический регистр, на который ссылалась команда, ставится в соответствие выделенному физическому регистру (табл. 9.5).

Таблица 9.5. Механизм переименования регистров

Команда	Действие	Рабочий регистр
i	Пишет в R0	С этого момента регистру R0 соответствует выделенный для команды регистр PHY0
i+1	Читает из R0	Читает из PHY0
i+2	Пишет в R0	С этого момента регистру R0 соответствует выделенный для команды регистр PHY1
i+3	Читает из R0	Читает из PHY1

При определении операндов команды имена логических регистров преобразуются в имена физических, после чего значения последних заносятся в поля операндов микрокоманд. Микрокоманды работают только с физическими регистрами.

Как можно увидеть, после декодирования команды i, которая в качестве приемника результата использует логический регистр R0, все прочие команды, использующие в качестве операнда R0, будут обращаться к физическому регистру, выделенному для команды i. При этом если какая-то команда после i будет писать в тот же логический регистр, ей будет выделен новый физический регистр, и все команды после нее будут использовать уже новый регистр.

Из табл. 9.5 видно, что команды стали независимы. Если команды, работающие с логическим регистром R0, зависят друг от друга и их нельзя выполнять параллельно, то микрокоманды "разведены" по физическим регистрам PHY0 и PHY1 и независимы.

Значение физического регистра переписывается в архитектурный, когда завершается выполнение команды (фиксируется ее результат). В свою очередь, завершение выполнения команды происходит, когда все предыдущие команды успешно завершились в заданном программой порядке.

Однако такой подход требует, чтобы микропроцессор помимо программно доступных архитектурных регистров содержал блок из гораздо большего количества невидимых программисту физических регистров, что и реализовано в большинстве современных микропроцессоров. Например, в микропроцессоре Itanium файл физических регистров имеет емкость 128 строк.

Как отмечалось выше, наличие конфликтов приводит к значительному снижению производительности микропроцессора. Определенные типы конфликтов требуют приостановки конвейера. При этом останавливается выполнение всех команд, находящихся на различных стадиях обработки (свыше 30 команд в Pentium 4). Другие конфликты, например, при неверном предсказанном направлении перехода, ведут к необходимости полной перезагрузки конвейера. Потери будут тем больше, чем более длинный конвейер используется в микропроцессоре. Такая ситуация явилась одной из причин сокращения числа ступеней в микропроцессорах последних моделей. Так, в микропроцессоре Itanium конвейер содержит всего 10 ступеней. При этом его тактовая частота составляет 1 ГГц.