15. Планирование кода

3.1 Постановка задачи

3.1.1 Цель планирования кода

- ◊ Цель планирования кода – выбрать такую последовательность команд, которая, *не меняя семантики программы*, обеспечит *близкое к оптимальному использование особенностей архитектуры целевого процессора,* и, как следствие, минимизирует общее время исполнения данной последовательности команд
- ◊ Прежде всего необходимо обеспечить использование возможностей параллельного выполнения команд, реализованных в аппаратуре

3.1 Постановка задачи

3.1.2 Сохранение семантики программы

- Требование сохранения семантики программы проще всего выразить в форме ограничений, которым должна удовлетворять целевая программа. Эти ограничения должны гарантировать, что оптимизированная программа будет давать такие же результаты, что и исходная.
- На планирование кода накладывается следующие три типа ограничений:
  - *Ограничения управления*. Все операции, выполняемые в исходной программе, должны выполняться и в оптимизированной программе.
  - *Ограничения данных.* Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе
  - *Ограничения ресурсов.* Планирование кода не должно требовать чрезмерного количества ресурсов.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.1 Конвейер обработки команд

	i	$i+1$	$i+2$	$i+3$	$i+4$
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

- ◊ Каждый современный процессор (ядро) использует конвейер (обработки) команд.
- ◊ При использовании конвейера выполнение каждой команды производится поэтапно.
- ◊ На рисунке показан простой пятиэтапный конвейер команд: на первом этапе выполняется выборка команды (**`IF`**), затем – ее декодирование (**`ID`**), выполнение (**`EX`**), обращение к памяти (**`MEM`**) и запись результата (**`WB`**).
- ◊ На рисунке показано, как одновременно могут выполняться команды i, i + 1, i + 2, i + 3 и i + 4.
  Каждая строка соответствует такту системных часов, а каждый столбец на рисунке определяет, на каком такте выполняются этапы каждой из команд.
  После разгона конвейера параллельно **может** выполняться пять команд.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.1 Конвейер обработки команд

- ◊ Большинство процессоров общего назначения динамически определяют зависимости между последовательными командами и автоматически приостанавливают выполнение команды в случае недоступности ее операндов.
- ◊ Некоторые процессоры, оставляют проверку зависимостей программному обеспечению по соображениям простоты и снижения энергопотребления процессора.
  В этом случае компилятор отвечает за вставку в код команд операций *NOP* («нет операции») там, где необходимо гарантировать доступность результатов предыдущей операции.
- ◊ Многие процессоры могут обрабатывать сразу несколько команд. Параллельное выполнение на таких машинах тоже может управляться как аппаратно, так и программно.
  Машины с программной обработкой параллельности известны под аббревиатурой *VLIW* (*Very Long Instruction Word*); машины с аппаратной обработкой известны под названием *суперскалярных*.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.2 Современные процессоры

- ◊ Процессоры *VLIW*, как следует из их названия, имеют команды большого размера, которые содержат несколько *операций*. Компилятор решает, какие операции будут выполняться параллельно, и явно указывает эту информацию в машинном коде.
- ◊ Суперскалярные процессоры автоматически обнаруживают зависимости между операциями и выполняют операции, как только их операнды становятся доступными.
- ◊ **Простые аппаратные планировщики** выполняют операции в том порядке, в котором выполняется их выборка.
  Если планировщик обнаруживает зависимую операцию, команда, содержащая эту операцию, и все последующие команды должны дождаться, пока зависимости не будут разрешены (то есть пока не станут доступны все требующиеся результаты выполнения предыдущих команд). Очевидно, что таким процессорам выгодна работа со статическим планировщиком, который размещает независимые операции одна за другой в порядке выполнения.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.2 Современные процессоры

- Процессоры *VLIW*, как следует из их названия, имеют команды большого размера, которые содержат несколько *операций*. Компилятор решает, какие операции будут выполняться параллельно, и явно указывает эту информацию в машинном коде.
- Суперскалярные процессоры автоматически обнаруживают зависимости между операциями и выполняют операции, как только их операнды становятся доступными.
- **Простые аппаратные планировщики** выполняют операции в том порядке, в котором выполняется их выборка.
  Если планиро[Поток команд приостанавливается]перацию, команда, содержащая эту операцию, и все последующие команды должны дождаться, пока зависимости не будут разрешены (то есть пока не станут доступны все требующиеся результаты выполнения предыдущих команд). Очевидно, что таким процессорам выгодна работа со статическим планировщиком, который размещает независимые операции одна за другой в порядке выполнения.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.2 Современные процессоры

◊ Более сложные планировщики могут выполнять команды «не по порядку» (*Out of Order*). Отдельные операции независимо приостанавливаются и не выполняются до тех пор, пока не будут получены все значения, от которых эти операции зависят.
Но статические планировщики могут помочь в работе даже столь интеллектуальным планировщикам, поскольку аппаратные планировщики ограничены объемом памяти, в котором могут храниться отложенные операции. Статический же планировщик может разместить независимые операции поближе одна к другой с тем, чтобы более эффективно использовать аппаратные возможности процессора.

◊ Следует отметить, что независимо от возможностей аппаратного планировщика, он не в состоянии работать с командами, которые еще не были выбраны.
Компилятор может повысить производительность динамического планировщика, обеспечивая возможность параллельного выполнения вновь выбираемых команд.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.2 Современные процессоры

◊ Более сложные планировщики могут выполнять команды «не по порядку» (*Out of Order*). Отдельные операции независимо приостанавливаются и не выполняются до тех пор, пока не будут получены в Поток команд не приостанавливается и зависят.
Но статические планировщики могут помочь в работе даже столь интеллектуальным планировщикам, поскольку аппаратные планировщики ограничены объемом памяти, в котором могут храниться отложенные операции. Статический же планировщик может разместить независимые операции поближе одна к другой с тем, чтобы более эффективно использовать аппаратные возможности процессора.

◊ Следует отметить, что независимо от возможностей аппаратного планировщика, он не в состоянии работать с командами, которые еще не были выбраны.
Компилятор может повысить производительность динамического планировщика, обеспечивая возможность параллельного выполнения вновь выбираемых команд.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.3 Параллелизм на уровне команд

- ◊ Параллелизм на уровне команд был впервые использован в компьютерной архитектуре как средство повышения скорости выполнения обычного последовательного кода.
- ◊ Компиляторы компьютеров, не содержащих средств параллельного выполнения команд, переупорядочивали команды таким образом, чтобы минимизировать количество используемых регистров.
- ◊ При применении таких компиляторов на новых компьютерах они, экономя регистры, одновременно минимизировали и возможности параллельного выполнения команд.
  В результате в последовательном коде оставалось очень мало возможностей для обеспечения параллелизма на уровне команд.
- ◊ Одной из первых мер по обеспечению параллельного выполнения команд (операций) стала возможность аппаратного переименования регистров для отмены результатов компилятора по оптимизации использования регистров.

3.2 Аппаратные возможности параллельного выполнения команд

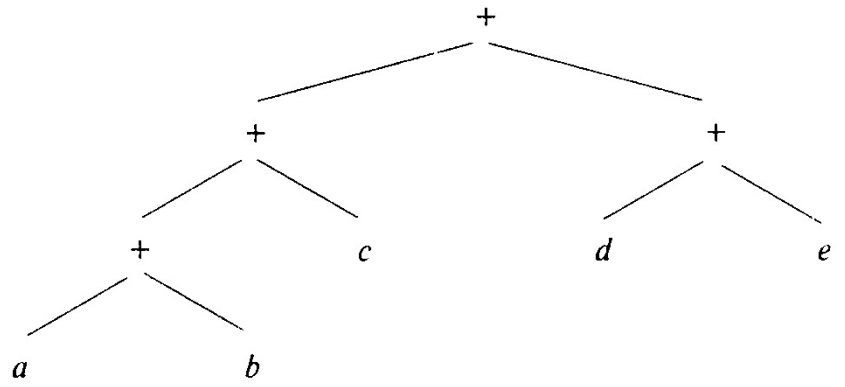
3.2.4 Аппаратное переименование регистров

- **Аппаратное переименование регистров** динамически изменяет назначение регистров в процессе выполнения программы.
  Оно интерпретирует машинный код, сохраняет значения, предназначенные для одного и того же регистра, в различных внутренних регистрах, и изменяет все их использования так, чтобы обращения к соответствующим регистрам выполнялись корректно.
- Поскольку, в первую очередь, ограничения, связанные с зависимостями регистров, возникают благодаря работе компилятора, они могут быть устранены путем использования алгоритма распределения регистров, который учитывает параллелизм на уровне команд.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.4 Аппаратное переименование регистров

◊ **Пример**. Вычисление выражения **(a + b) + c + (d + e)** (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):




```
LD  r1, a          // r1 = a
LD  r2, b          // r2 = b
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, c          // r2 = c
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, d          // r2 = d
LD  r3, e          // r3 = e
ADD r2, r2, r3     // r2 = r2+r3
ADD r1, r1, r2     // r1 = r1+r2
```

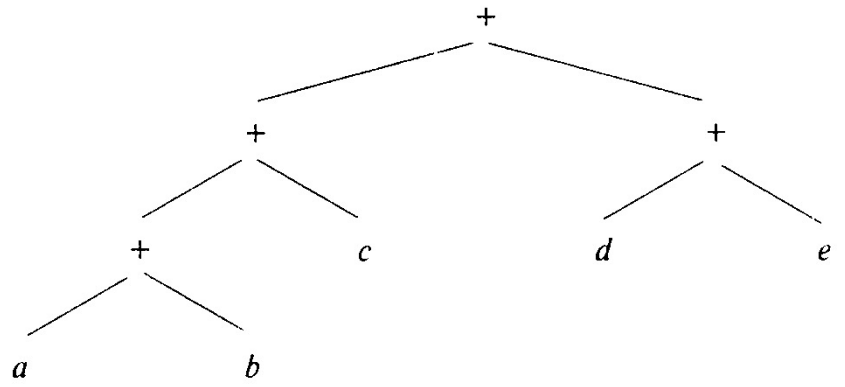
◊ Как видно из текста процедуры, для этого регистры **r1** и **r2** необходимо использовать несколько раз. Повторное использование регистров требует последовательного вычисления.

◊ Единственные операции, которые могут быть выполнены параллельно: загрузки значений из ячеек **a** и **b** и загрузки значений из ячеек **d** и **e**. Таким образом, всего для полного вычисления выражения с максимальным использованием параллельности требуется 7 шагов вместо 9.

3.2 Аппаратные возможности параллельного выполнения команд

3.2.4 Аппаратное переименование регистров

◊ **Пример**. Вычисление выражения **(a + b) + c + (d + e)** (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):




```
LD  r1, a          // r1 = a
LD  r2, b          // r2 = b
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, c          // r2 = c
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, d          // r2 = d
LD  r3, e          // r3 = e
ADD r2, r2, r3     // r2 = r2+r3
ADD r1, r1, r2     // r1 = r1+r2
```

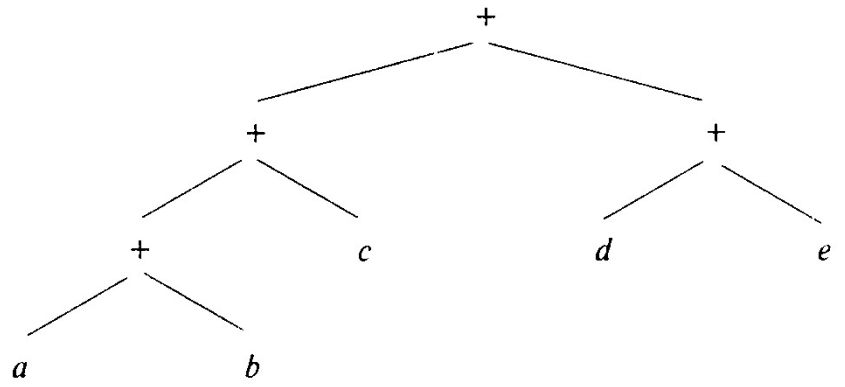
◊ Если использовать различные регистры для каждой промежуточной суммы (то есть **использовать не 3, а 5 регистров**), то выражение можно вычислить всего за 4 шага, что равно высоте дерева выражения.

r1 = a	r2 = b	r3 = c	r4 = d	r5 = e
r6 = r1+r2	r7 = r4+r5			
r8 = r6+r3				
r9 = r8+r7				

3.2 Аппаратные возможности параллельного выполнения команд

3.2.4 Аппаратное переименование регистров

◊ **Пример**. Вычисление выражения **`(a + b) + c + (d + e)`** (см. дерево на рисунке) можно выполнить с использованием трех регистров (числа Ершова):




```
LD  r1, a          // r1 = a
LD  r2, b          // r2 = b
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, c          // r2 = c
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, d          // r2 = d
LD  r3, e          // r3 = e
ADD r2, r2, r3     // r2 = r2+r3
ADD r1, r1, r2     // r1 = r1+r2
```

◊ Регистр **`r1`** переименовывается в **`r3`**,
`r2` – в **`r4`**, **`r3`** – в **`r5`**
с помощью специальных команд процессора

…ждой промежуточной
…**ров**), то выражение
…соте дерева выражения.

`r1 = a`	`r2 = b`	`r3 = c`	`r4 = d`	`r5 = e`
`r6 = r1+r2`	`r7 = r4+r5`			
`r8 = r6+r3`				
`r9 = r8+r7`				

3.3 Модель процессора

3.3.1 Базовая модель процессора

- Многие процессоры можно описать, используя следующую модель: процессор представляется парой $P = \langle R, T \rangle$,
  - T – множество типов операций, таких как загрузки (на регистры), сохранения (в память), арифметические операции и т.п.;
  - $R = [r_1, r_2, ...]$ – вектор, представляющий аппаратные ресурсы: r_i – количество доступных единиц i-го ресурса.
- **Примеры аппаратных ресурсов:**
  - модуль обращения к памяти (МОП),
  - арифметико-логическое устройство (АЛУ),
  - функциональное устройство (ФУ),
  - модуль для работы с числами с плавающей точкой (МПТ).
- **Примеры функциональных устройств**: сумматор, умножитель, …

3.3 Модель процессора

3.3.1 Базовая модель процессора

- *Команда* может содержать одну или более *операций*.
  Для начала выполнения операции необходимо наличие:
  - входных операндов,
  - ресурсов, требующихся для выполнения операции.

- С каждым *входным операндом* связана *входная задержка*, указывающая, через сколько тактов после начала выполнения операции должно быть доступно его значение.
  Типичные входные операнды имеют нулевую задержку, означающую, что их значения нужны в момент начала выполнения операции.

- С каждым *выходным операндом* связана *выходная задержка*, указывающая, через сколько тактов после начала выполнения операции становится доступен *результат*.

3.4 Анализ зависимостей

3.4.1 Зависимости по данным

◊ **Определение**. Две команды называются *зависимыми по данным*, если изменение порядка их выполнения может привести к изменению результата вычислений, выполняемых программой.

◊ Виды зависимостей:

- ◈ *Истинная зависимость*: **чтение после записи**. Если команда C_1 записывает значение в некоторую ячейку памяти, а команда C_2 считывает это значение, то команды C_1 и C_2 зависимы.
- ◈ *Антизависимость*: **запись после чтения**. Если команда C_1 считывает значение из некоторой ячейки памяти, а команда C_2 записывает в эту ячейку новое значение, то команды C_1 и C_2 зависимы.
- ◈ *Зависимость по выходу*: **запись после записи**. Если команды C_1 и C_2 записывают значения в одну и ту же ячейку памяти, то команды C_1 и C_2 зависимы.

3.4 Анализ зависимостей

3.4.1 Зависимости по данным

◊ **Замечание**. ***Неустранимой является только истинная зависимость***, а два других вида зависимостей (их называют *зависимостями, связанными с хранением*) могут быть устранены путем использования в командах C_1 и C_2 разных ячеек памяти для разных значений.

◊ В общем случае задача выявления зависимостей алгоритмически неразрешима: компилятор обязан предполагать, что две команды могут обращаться к одним и тем же ячейкам (с учетом указателей), если он не может установить обратное (консервативность).

3.4 Анализ зависимостей

3.4.1 Зависимости по данным

◊ **Пример**: Рассмотрим код

```
1. a  = 1
2. *p = 2
3. b  = a
```

В этом коде сразу можно обнаружить истинную зависимость между командами 1 и 3. Больше зависимостей как будто нет, но это только в том случае, когда *компилятор может установить, что указатель* **p** *не может указывать на* **a**. Если это не так, компилятор обязан считать, что указатель **p** может указывать на **a**, и тогда возникают еще две зависимости:

- ◈ истинная зависимость между командами 2 и 3
- ◈ зависимость по записи между командами 1 и 2.

◊ Пример показывает, что выявление зависимостей связано с *межпроцедурным анализом алиасов*.

3.4 Анализ зависимостей

3.4.2 Зависимости по данным между итерациями цикла

◊ Анализ зависимостей по данным для массива должен выяснить, анализируя индексные выражения обращений к массиву, могут ли два обращения ссылаться на один и тот же элемент массива при каких-нибудь значениях индексных переменных.

◊ **Пример**: Рассмотрим цикл

```
for(i=0; i<n; ++i)
    A[2*i] = A[2*i+1];
```

Обращения к массиву **`A`** в левой и правой частях оператора присваивания независимы, так как ни при каком значении индекса **`i`** значение индекса в левой части (четное число) не станет равным значению индекса в правой части (нечетное число).

Пример не должен вводить в заблуждение: анализ зависимостей по данным для итераций цикла, содержащих обращения к элементам массивов – сложная задача.

3.4 Анализ зависимостей

3.4.2 Зависимости по данным между итерациями цикла

◊ В случае, когда все индексы массивов и все границы циклов являются аффинными выражениями задача сводится к задаче целочисленного линейного программирования.

3.4 Анализ зависимостей

3.4.3 Зависимости по управлению

◊ Команда **`i1`** называется зависящей по управлению от команды **`i2`**, если выполнение команды **`i2`** определяет, будет ли выполняться команда **`i1`**.

Простые примеры

- в конструкции **`if(c) s1; else s2;`** **`s1`** и **`s2`** зависят по управлению от команды проверки условия **`c`**
- в конструкции **`while (c) s;`** **`s`** зависит по управлению от команды проверки условия **`c`**

3.4 Анализ зависимостей

3.4.2 Зависимости по управлению

- Команда **`i1`** называется зависящей по управлению от команды **`i2`**, если выполнение команды **`i2`** определяет, будет ли выполняться команда **`i1`**.
- **Пример**. Во фрагменте кода

```
if (a > t)
    b = a * a;
d = a + c;
```

  - команды, соответствующие инструкциям **`b = a * a`** и **`d = a + c`**, не зависят по данным от других частей фрагмента.
  - команда, соответствующая инструкции **`b = a*a`**, зависит по управлению от сравнения **`a > t`**
  - команда, соответствующая инструкции **`d = a+c`**, не зависит от сравнения **`a > t`** ни по данным, ни по управлению

3.5 Планирование базовых блоков

3.5.1 Граф зависимостей по данным

- Граф зависимостей по данным $G = (N, E)$, содержит
  - множество узлов N, представляющих операции в командах базового блока,
  - множество ориентированных ребер E, представляющих зависимости по данным между операциями.
- Граф зависимостей по данным отражает все зависимости по данным, имеющиеся в блоке.

Примеры расписания выполнения команд на конвейере

- **ADD r1, r2, r3**
  - r2, r3: требуются перед E2
  - r1: готов после E2
- **MOV r1, r2 asl #const**
  - r2: требуется перед E1
  - r1: готов после E1
- Такие две команды могут начать выполняться на одном такте:

	E1	E2
`mov r1, r2 << 2`	R2 R1	
`add r2, r1, r2`		R1, R2 R2

← *Стадии конвейера*

- При выдаче этих команд в обратном порядке появится задержка в 2 такта:

	E1	E2	E3
`add r2, r1, r2`	--	R1, R2 R2	
`mov r1, r2 << 2`	Ожидание готовности R2		R2 R1

Обозначения:
Rn – Регистр требуется в начале стадии выполнения Ei
Rk – Записывается в конце стадии Ei

3.5 Планирование базовых блоков

3.5.1 Граф зависимостей по данным

◊ Узлы и ребра графа G строятся следующим образом:

- Каждой операции $n \in N$ ставится в соответствие таблица резервирования ресурсов (RT_n), определяемая в соответствии с типом операции n.
  Каждое ребро $e \in E$ помечается *задержкой* d_e, указывающей, что целевой узел может быть выполнен не ранее, чем через d_e тактов после выполнения исходного узла.
- Пусть за операцией n_1 следует операция n_2, причем обе обращаются к одной и той же ячейке памяти с запаздываниями l_1 и l_2 соответственно
  (значение в ячейке создается через l_1 тактов после начала первой команды и требуется второй команде через l_2 тактов после ее начала: типичными являются значения $l_1 = 1$ и $l_2 = 0$).
  Тогда в множество E необходимо включить ребро $n_1 \rightarrow n_2$, помеченное задержкой $l_1 - l_2$.

3.5 Планирование базовых блоков

3.5.1 Граф зависимостей по данным

◊ **Пример**: Для кода

```
1. LD  R1, #0
2. ADD R1, R1, R1
3. LD  R2, #8
4. MUL R1, R1, R2
5. LD  R2, #16
6. MUL R1, R1, R2
7. LD  R2, #24
8. MUL R1, R1, R2
9. ST  ..., R1
```

(цифры означают номера команд, все зависимости – истинные)

граф зависимостей по данным показан на рисунке

◊ Самый длинный путь в графе зависимостей называется *критическим.* В графе, показанном на рисунке, критическим является путь 1-2-4-6-8-9.

◊ Построение графа зависимостей состоит в обходе программы снизу вверх, построении узла для каждого нового значения и соединения этого узла со всеми узлами, использующими соответствующее значение.

3.5 Планирование базовых блоков

3.5.2 Пример

- Рассмотрим простой процессор (ядро), который может выполнять в каждый момент времени две операции:
  - первая операция должна быть
    - либо операцией АЛУ **`OP dst, src1, src2,`**
    - либо операцией ветвления
  - вторая операция должна быть
    - либо операцией загрузки **`LD dst, addr`**
    - либо операцией сохранения **`ST addr, src`**

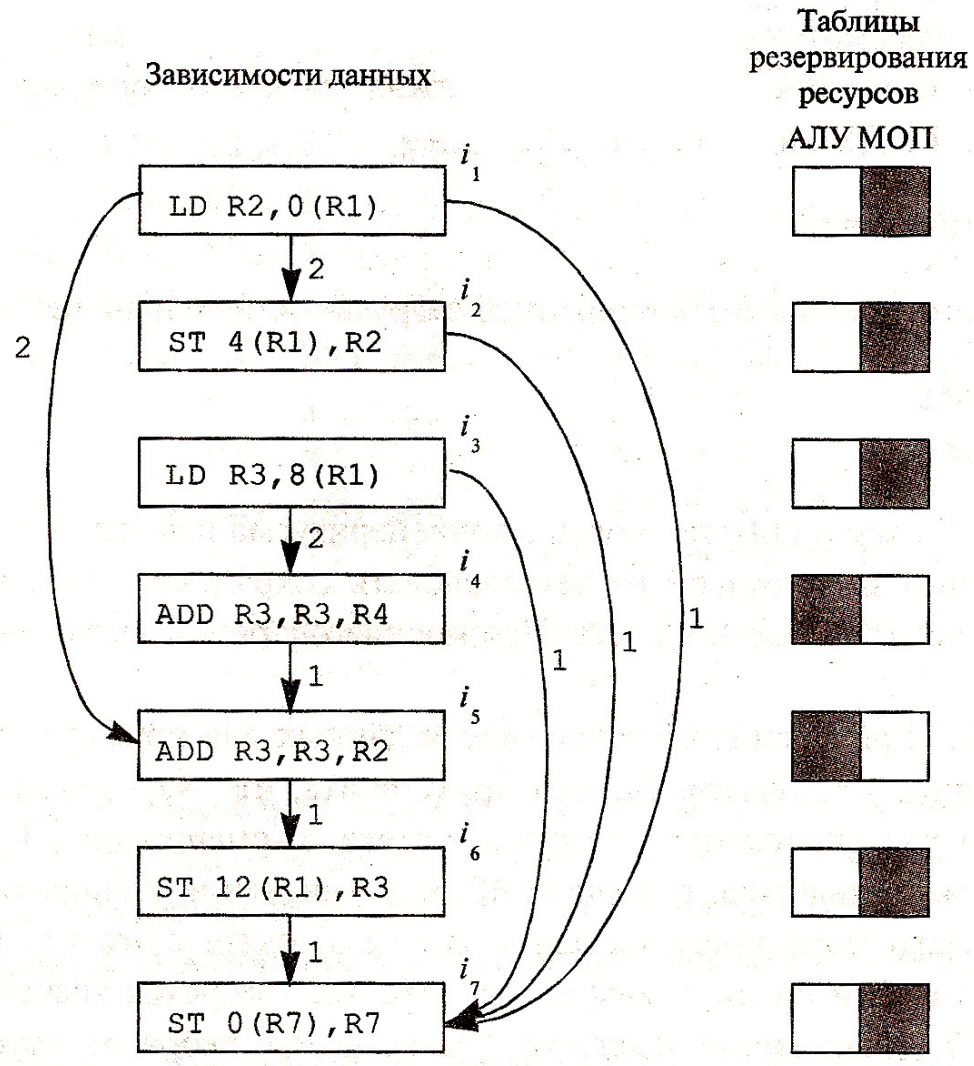
- Операции загрузки (**`LD`**) и сохранения (**`ST`**) полностью конвейеризуемы и занимают два такта. Все прочие операции выполняются за один такт.

- Пусть **`R1`** – указатель стека данных, причем **`d(R1)`** обозначает адрес по стеку со смещением **`d`** относительно **`R1`**, используемый для обращения к данным в стеке со смещениями 0 и 12 (в примере на следующем слайде).

3.5 Планирование базовых блоков

3.5.2 Пример

- ◊ Еще один пример графа зависимостей по данным.
- ◊ Таблицу резервирования ресурсов удобно изображать в виде сетки с закрашенными и пустыми ячейками. Каждый столбец соответствует одному из ресурсов, а каждая строка – одному из тактов выполнения операции.
  Пусть никакая операция не требует более одной единицы одного ресурса. Тогда можно представить единицы закрашенными квадратами, а нули – пустыми.



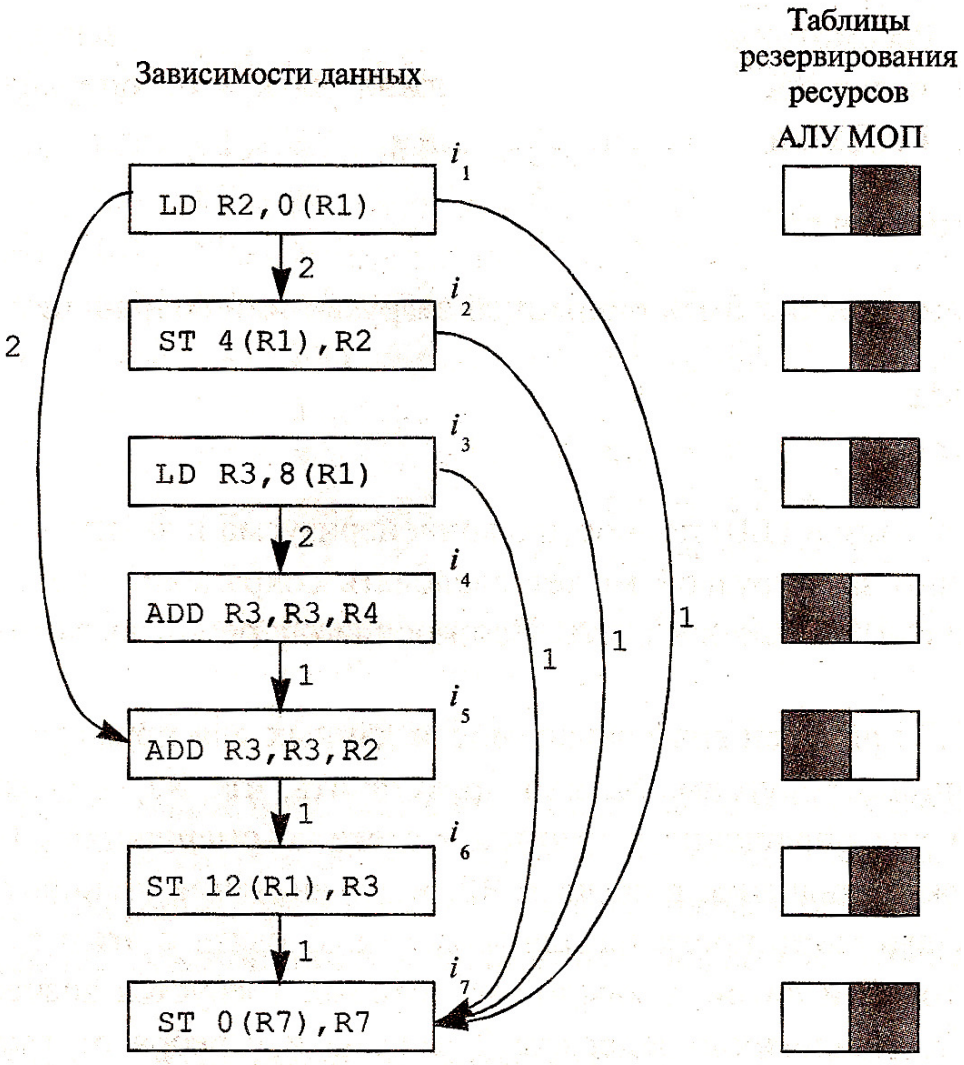
3.5 Планирование базовых блоков

3.5.2 Пример

Загрузка регистра занимает 2 такта; следовательно, операция, одним из операндов которой является регистр, загружаемый командой (операцией) LD, может начаться не раньше, чем через 2 такта после начала операции загрузки регистра LD.

из ресурсов, а каждая строка – одному из тактов выполнения операции.

Пусть никакая операция не требует более одной единицы одного ресурса. Тогда можно представить единицы закрашенными квадратами, а нули – пустыми.

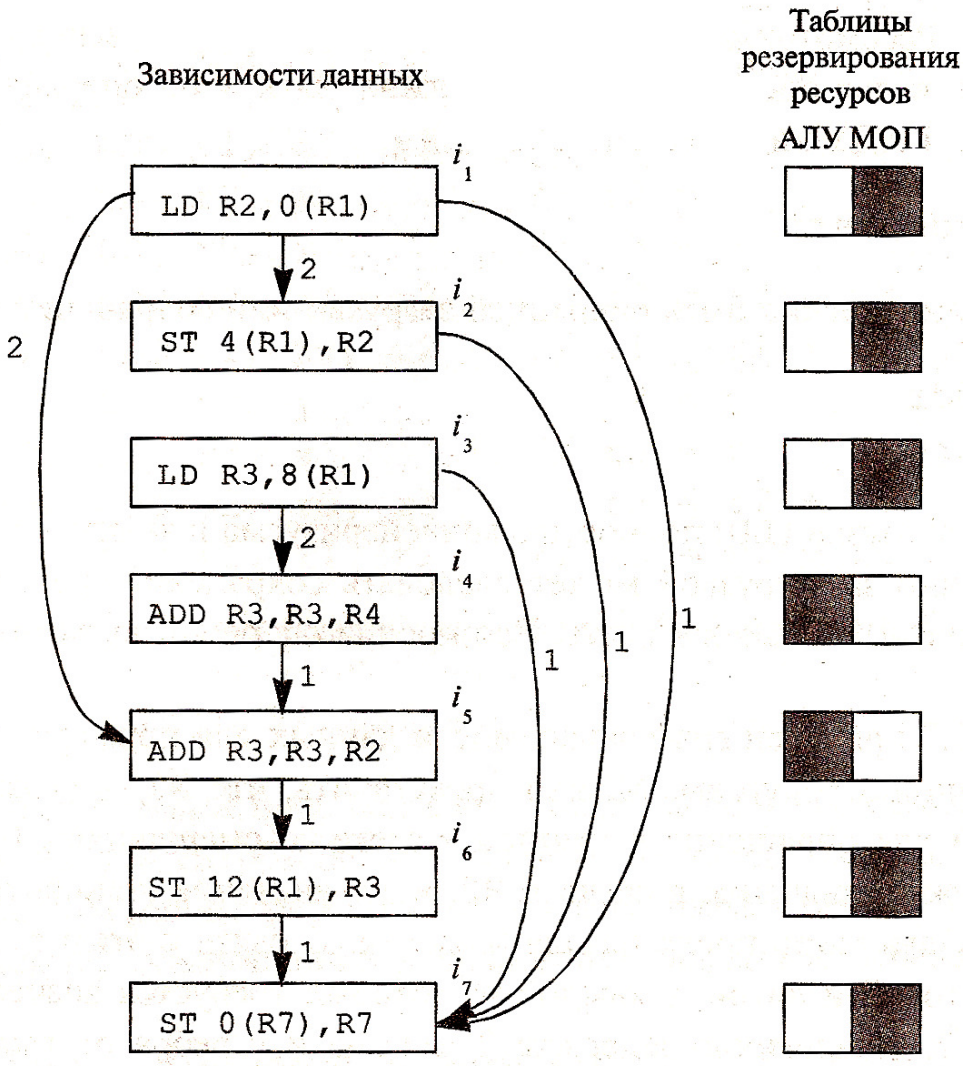


3.5 Планирование базовых блоков

3.5.2 Пример

Загрузка регистра занимает 2 такта; следовательно, операция, одним из операндов которой является регистр, загружаемый командой (операцией) LD, может начаться не раньше, чем через 2 такта после начала операции загрузки регистра LD.

Арифметическая операция (например, ADD) занимает 1 такт; следовательно, операция ST сохранения ее результата, может начаться через один такт; операции LD и ST, имеющие разные операнды (регистры и адреса памяти) независимы и могут выдаваться на каждом такте.



Планировщик и алиасы указателей

```
void copy(char *p, char *q, size_t N) {
// void copy(char * restrict p, char * restrict q, size_t N) {
//  assert(N%4 == 0);
  for (int i=0; i<N/4; i+=4) {
    p[i] = q[i];
    p[i+1] = q[i+1];
    p[i+2] = q[i+2];
    p[i+3] = q[i+3];
  }
}
// godbolt: https://godbolt.org/z/a7ehx4chG
p = a + 1;
q = a;

a: [ 0 1 2 3 4 5 6 7 8 ]
p:     ^p[i] == q[i+1]
q:   ^q[i]
```

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

- ◊ Алгоритм планирования списка работает без откатов: каждый узел планируется раз и только раз.
- ◊ Простейший подход к планированию базового блока включает посещение каждого узла графа в «приоритетном топологическом порядке»
- ◊ Граф зависимостей по данным, будучи ориентированным ациклическим графом, имеет минимум один, а на самом деле – несколько топологических порядков. Наиболее удобный из них называется «приоритетным топологическим порядком»

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

- Приоритетные топологические порядки:
  - При отсутствии ограничений, связанных с ресурсами, кратчайший план получается при помощи критического пути – самого длинного пути в графе зависимости данных. Метрика, используемая в качестве функции приоритета – высота узла, представляющая собой длину самого длинного пути от данного узла в графе.
  - Если все операции независимы (между ними нет зависимостей по данным и/или по управлению), то длина плана ограничена доступными ресурсами. Критический ресурс – это ресурс с наибольшим отношением количества использований к количеству доступных единиц ресурса. Более высокий приоритет будут иметь операции, использующие более критические ресурсы.
  - Для разрешения неопределенностей можно использовать исходный порядок операций – операция, находящаяся в исходной программе раньше других, должна быть запланирована раньше.

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

- ◊ Алгоритм обходит узлы графа зависимостей по данным в «приоритетном топологическом порядке» и для каждого узла вычисляет наиболее раннее время, когда он может быть вычислен, в соответствии с ограничениями зависимости данных ранее спланированных узлов.
- ◊ Затем выполняется проверка наличия требующихся для узла ресурсов в таблице резервирования ресурсов, в которой собраны все выделенные к настоящему времени ресурсы, и план корректируется: команда планируется к выполнению в самый ранний момент, в который имеется достаточно ресурсов.

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

Вход: вектор ресурсов машины $R = [r_1, r_2, ...]$, граф зависимостей по данным $G = (N, E)$; каждой операции $n \in N$ сопоставлена таблица резервирования ресурсов RT_n; каждое ребро $e = n_1 \rightarrow n_2$ из E помечено задержкой d_e: n_2 можно выполнить не ранее чем через d_e тактов после n_1.

Выход: план S, который отображает операции из N на временны́е интервалы, когда удовлетворяются все ограничения по ресурсам для этих операций и когда может начинаться их выполнение.

Метод: выполнить программу (псевдокод):

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

```
RT = пустая таблица резервирования ресурсов;
for (каждый n ∈ N в приоритетном
топологическом порядке) {
/* Вычисление самого раннего времени S(n) = s
   возможного начала команды n на основе заданного
   S(p)- времени начала предыдущей команды p*/
        s = max(e = p→n)∈E (S(p) + de);
/* Вычисление задержки начала выполнения команды n
   до тех пор, пока не будут доступны все ресурсы (i).*/
        while(∃ i: RT[s][i]+ RTn[i] > R[i])
                s = s + 1;
        S(n) = s;
/* Корректировка таблицы ресурсов для учета
   ресурсов, используемых командой n */
        for (всех ресурсов i)
                RT[s][i] += RTn[i]
}
```

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования сп

```
RT = пустая таблица резе
for (каждый n ∈ N в прис
топологическом порядке)
/* Вычисление самого ран
   возможного начала ком
   S(p)- времени начала
        s = max(e = p→n)∈E(S(p) + de);
/* Вычисление задержки начала выполнения команды n
   до тех пор, пока не будут доступны все ресурсы (i).*/
     while(∃ i: RT[s][i]+ RTn[i] > R[i])
            s = s + 1;
     S(n) = s;
/* Корректировка таблицы ресурсов для учета
   ресурсов, используемых командой n */
        for (всех ресурсов i)
              RT[s][i] += RTn[i]
}
```

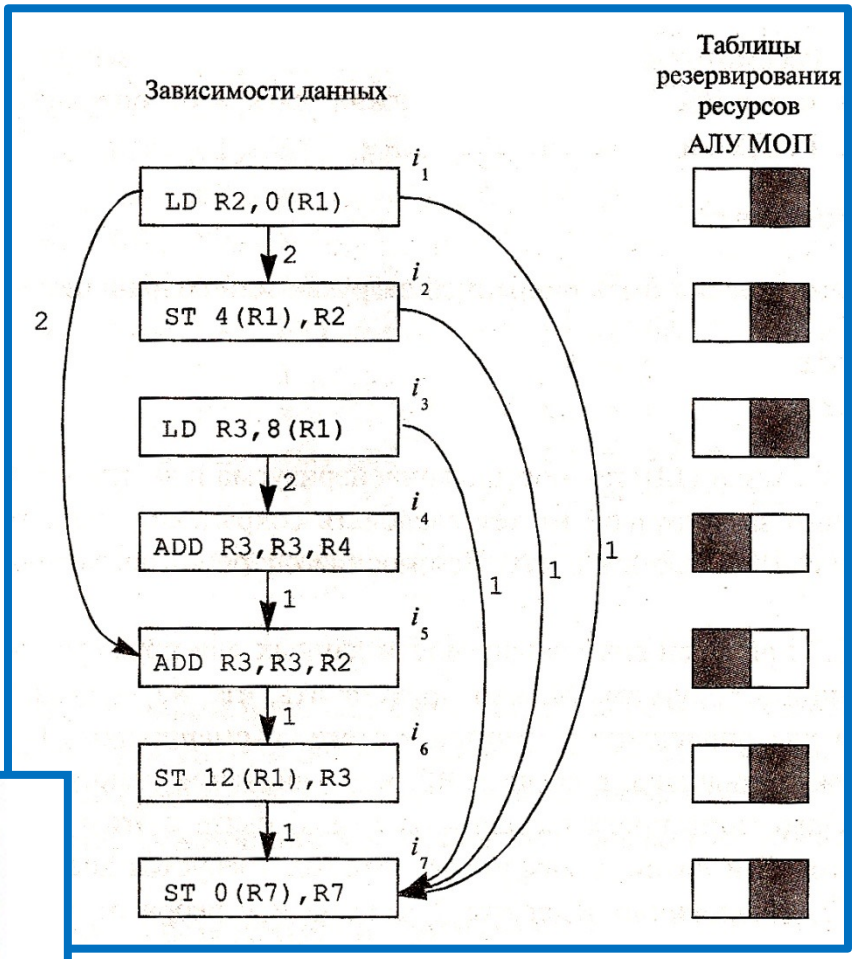
while: увеличиваем такт, на котором будет выдана инструкция, пока для некоторого *i* кол-во ресурсов *i*-го типа на текущем такте *s* `RT[s][i]` + кол-во ресурсов требуемых текущей инструкцией $RT_n[i]$ все еще > общего имеющегося кол-ва ресурсов i-го типа `R[i]`

3.5 Планирование базовых блоков

3.5.3 Алгоритм планирования списка

Пример

- В графе зависимости данных (справа) критический путь имеет длину 6 тактов: путь от **`LD R3,8(R1)`** до **`ST 0(R7),R7`**. Суммарные задержки вдоль ребер равны 5, и к ним добавляется еще один такт, необходимый для выполнения команды **`ST 0(R7),R7`**. Оптимальный план, выданный алгоритмом, показан на нижнем рисунке



	АЛУ МОП
LD R3,8(R1)	
LD R2,0(R1)	
ADD R3, R3,R4	
ADD R3, R3,R2	ST 4(R1),R2
	ST 12(R1),R3
	ST 0(R7),R7

3.6 Глобальное планирование кода

3.6.1 Вводные замечания

◊ Для современных процессоров планы, создаваемые путем работы только с базовыми блоками, как правило, несовершенны и приводят к простаиванию большого количества ресурсов. Для более эффективного использования машинных ресурсов можно рассмотреть стратегии планирования, использующие перемещение команд между базовыми блоками. Стратегии планирования кода, которые затрагивают больше одного базового блока, называются стратегиями *глобального планирования.*

Стратегии глобального планирования должны гарантировать:

- оптимизированная программа выполняет все команды исходной программы;
- при выполнении некоторых команд исходной программы с упреждением не возникает никаких нежелательных побочных действий.

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

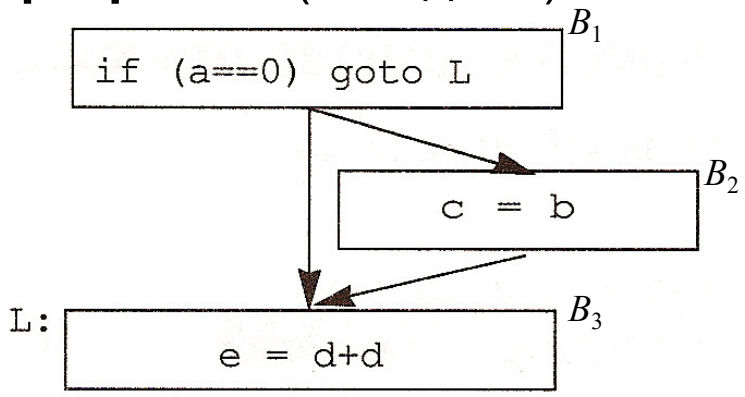
- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

- **Программа** (исходная):



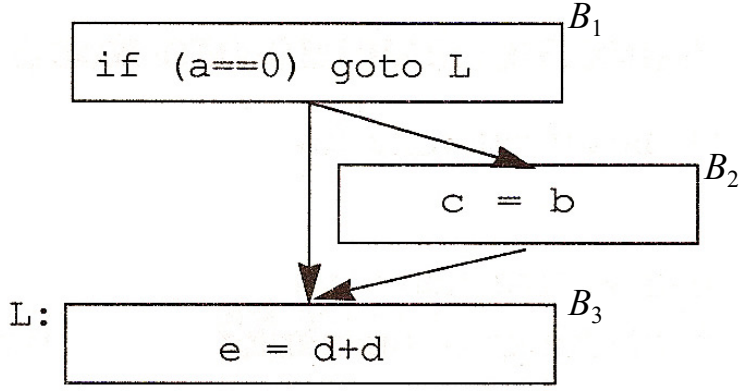
пусть все обращения к памяти корректны и удовлетворяются кэшем.

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

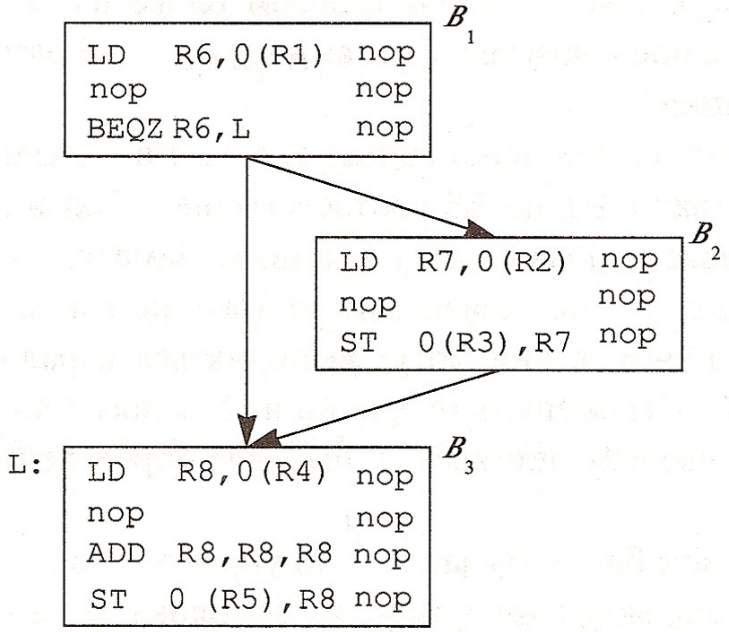
- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

- **Программа** (исходная):



пусть все обращения к памяти корректны и удовлетворяются кэшем.

Та же программа в машинном коде после локального планирования

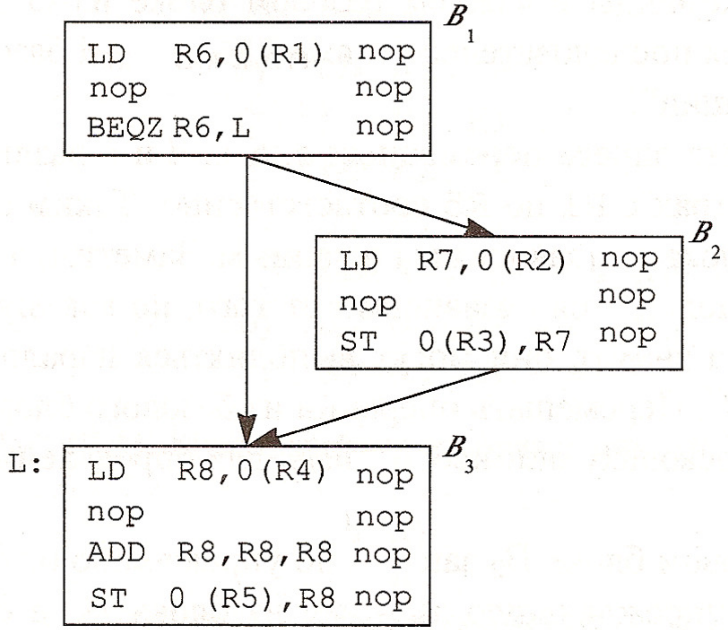


3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

Программа в машинном коде после локального планирования



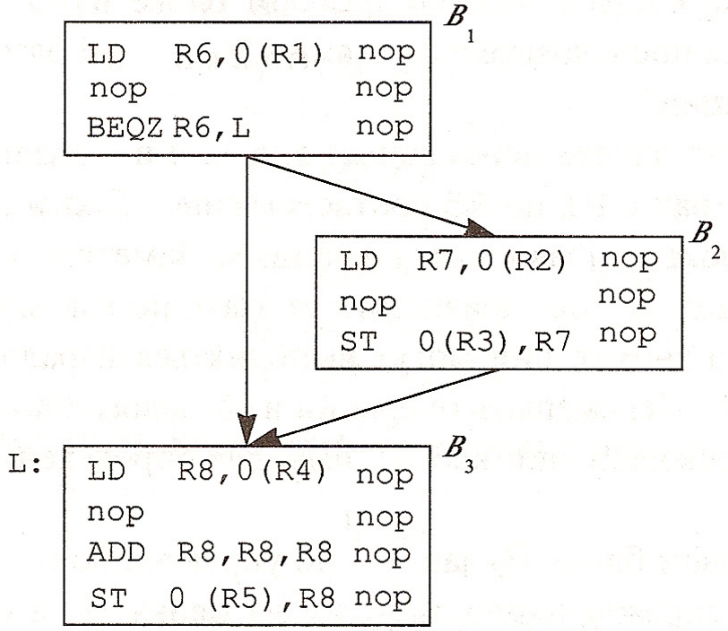
Команды в каждом блоке из-за зависимостей по данным должны выполняться последовательно, поэтому при локальном планировании в каждую команду была вставлена операция **nop**.

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

Программа в машинном коде после локального планирования



Команды в каждом блоке из-за зависимостей по данным должны выполняться последовательно, поэтому при локальном планировании в каждую команду была вставлена операция **nop**.

Пусть переменные a, b, c, d и e расположены в разных ячейках, а их адреса находятся на регистрах с **R1** по **R5** соответственно. Тогда у вычислений в разных базовых блоках нет зависимостей по данным, но есть зависимости по управлению.

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

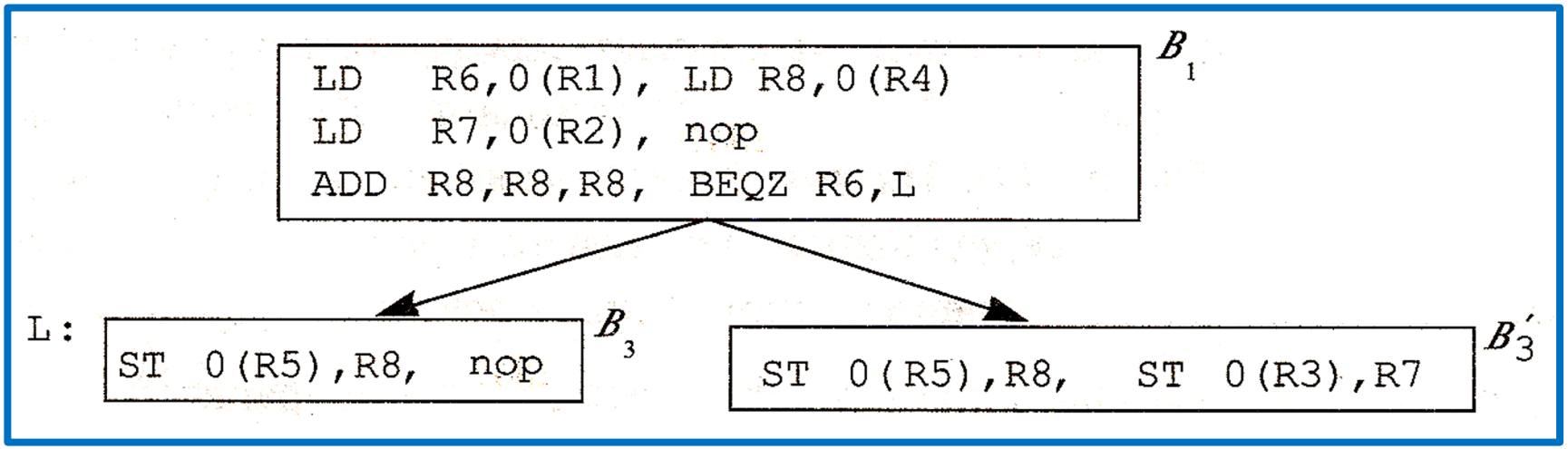
- **Процессор**
  - может выполнить любые две команды за один такт
  - каждая команда (кроме команды загрузки **LD**) выполняется с задержкой в один такт
  - команда загрузки выполняется с задержкой в два такта

- **Глобальное планирование**:
  - Все команды в блоке B_3 выполняются независимо от ветвления. Значит, они могут выполняться параллельно с командами из блока B_1.
  - Перемещать команды из блока B_1 в блок B_3 нельзя из-за зависимостей по управлению: они необходимы для выбора пути ветвления.
  - Команды в блоке B_2 зависят по управлению от проверки в блоке B_1. Можно выполнить загрузку из блока B_2 **с упреждением**, перенеся ее в блок B_1 (на этом экономится 2 такта).
  - сохранения выполнять с упреждением нельзя, так как необходимость сохранений выясняется только после ветвления, однако команды сохранения можно отложить (перенеся сохранение c из B_2 в копию B_3).

3.6 Глобальное планирование кода

3.6.2 Пример глобального планирования

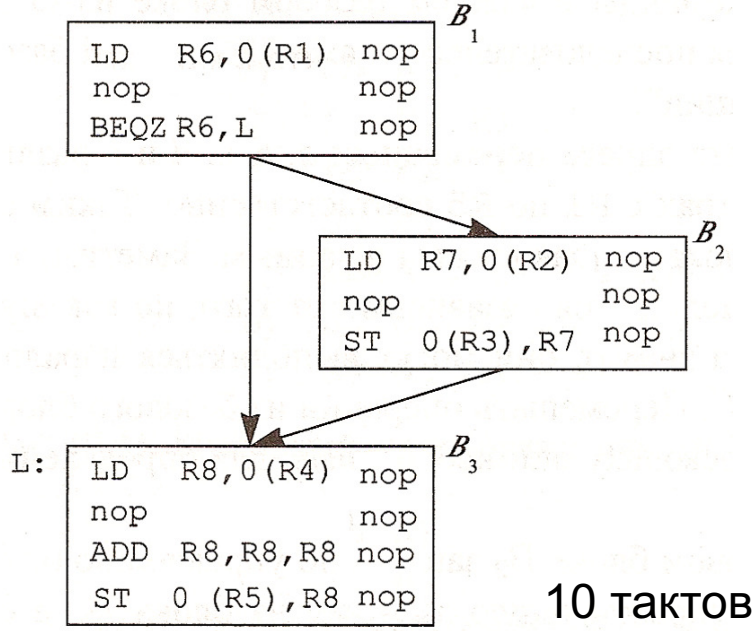
- **Глобальное планирование**:
  - Все команды в блоке B_3 выполняются независимо от ветвления. Значит, они могут выполняться параллельно с командами из блока B_1.
  - Перемещать команды из блока B_1 в блок B_3 нельзя из-за зависимостей по управлению: они необходимы для выбора пути ветвления.
  - Команды в блоке B_2 зависят по управлению от проверки в блоке B_1. Можно выполнить загрузку из блока B_2 **с упреждением**, перенеся ее в блок B_1 (на этом экономится 2 такта).
  - сохранения выполнять с упреждением нельзя, так как необходимость сохранений выясняется только после ветвления, однако команды сохранения можно отложить (перенеся сохранение c из B_2 в копию B_3).



3.6 Глобальное планирование кода

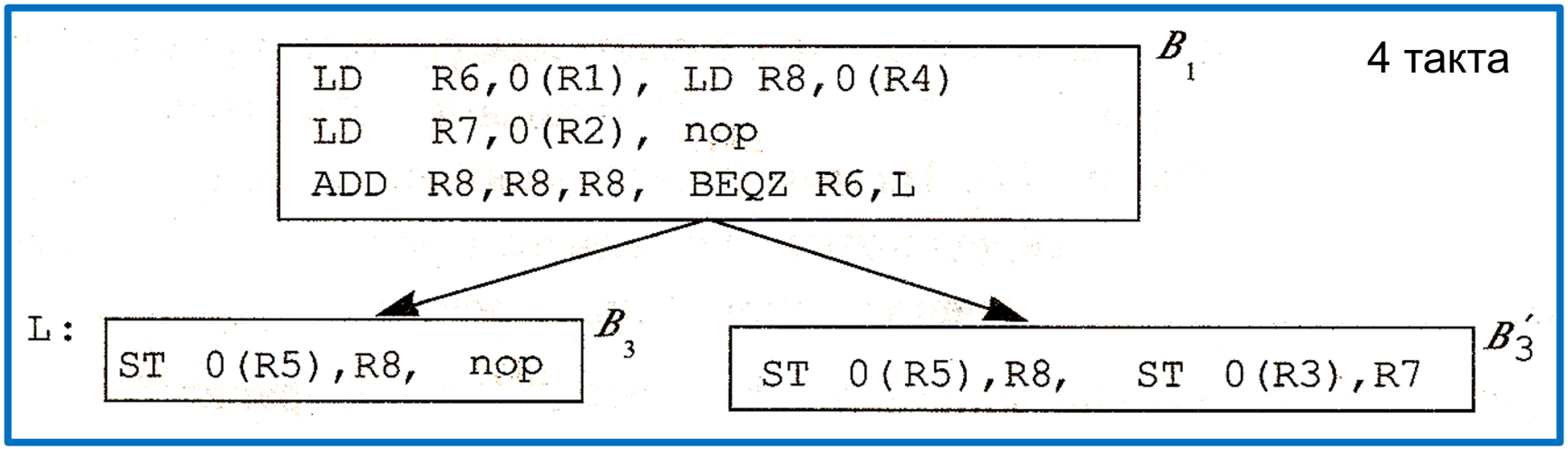
3.6.2 Пример глобального планирования

После локального планирования



10 тактов

После глобального планирования



3.6 Глобальное планирование кода

3.6.3 Эквивалентность по управлению

◊ **Определение**. Базовые блоки B и B' *эквивалентны по управлению*, если блок B выполняется тогда и только тогда, когда выполняется блок B'.

◊ **Утверждение**. Для того, чтобы базовые блоки B и B' были эквивалентны по управлению необходимо и достаточно, чтобы $B = Dom(B') \;\&\; B' = PostDom(B)$

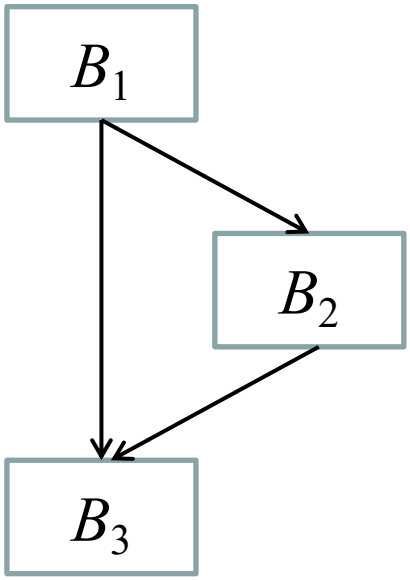
◊ **Пример**. В примере 3.6.2 блоки B_1 и B_3 эквивалентны по управлению.

3.6 Глобальное планирование кода

3.6.4 Перемещение кода

◊ Глобальное планирование кода между базовыми блоками. При этом, как видно из рассмотренного примера 3.6.2 возможны следующие отношения между блоками:

- Блоки B_1 и B_3 эквивалентны по управлению:
$$B_1 = Dom(B_3)\ \&\ B_3 = Postdom\ (B_1)$$



- Блок B_1 доминирует над блоком B_2,
а блок B_2 не постдоминирует над блоком B_1:
$$B_1 = Dom(B_2)\ \&\ B_2 \neq Postdom\ (B_1)$$

- Блок B_2 не доминирует над блоком B_3,
блок B_3 постдоминирует над блоком B_2:
$$B_2 \neq Dom(B_3)\ \&\ B_3 = Postdom\ (B_2)$$

- Пара блоков вдоль пути выполнения не связана ни отношением доминирования, ни отношением постдоминирования.

3.6 Глобальное планирование кода

3.6.5 Перемещение кода вверх по пути управления

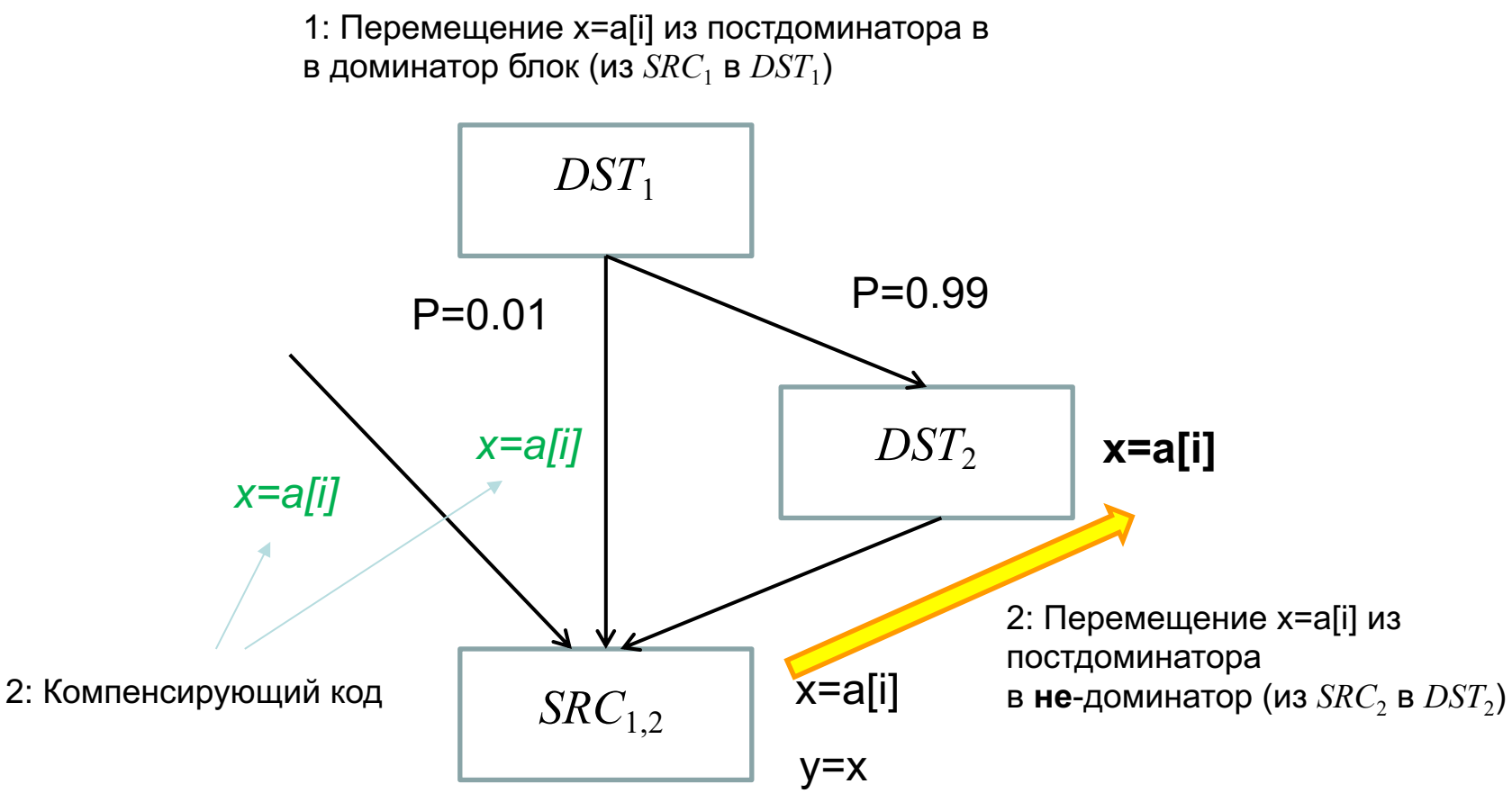
◊ Пусть команда из блока *src* перемещается вверх по пути управления в блок *dst*, не нарушая никаких зависимостей по данным.

1. $dst = Dom(src) \ \& \ src = Postdom(dst)$:
перемещенная команда выполняется только один раз

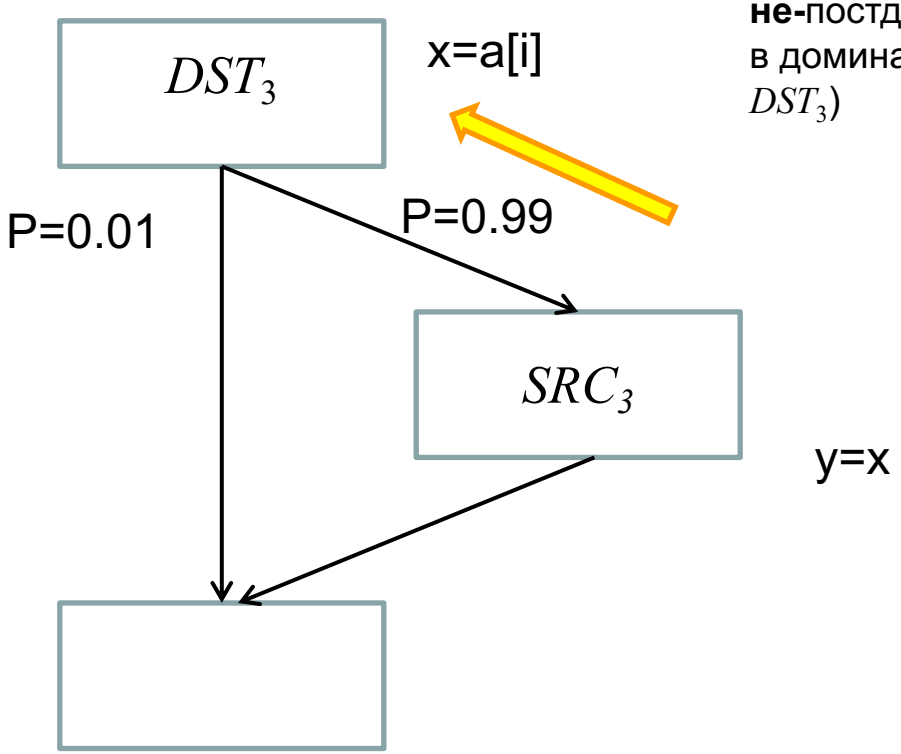
2. $dst \neq Dom(src) \ \& \ src = Postdom(dst)$:
существует один или несколько путей, которые достигают *src,* но не проходят через *dst;* в этом случае в базовые блоки, которые образуют разрез, отделяющий *Entry* от *src* необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях, достигающих *src*).

3. $dst = Dom(src) \ \& \ src \neq Postdom(dst)$:
существует один или несколько путей, которые проходит через *dst,* но не достигают *src,* и в результате на этих путях могут быть выполнены лишние команды; эти команды не должны приводить к нежелательным побочным эффектам, иначе перемещение кода **некорректно**

Перемещение кода вверх



Перемещение кода вверх



2: Перемещение x=a[i] из **не-**постдоминатора в доминатор (из SRC_3 в DST_3)

3.6 Глобальное планирование кода

3.6.5 Перемещение кода вверх по пути управления

- Перемещение кода повышает эффективность программы только тогда, когда оно использует только те ресурсы, которые иначе простаивали бы; такое перемещение называется *бесплатным*
- В каждом месте разреза, в которое вставляется команда, должны удовлетворяться следующие ограничения:
  - операнды вставленной команды должны иметь те же значения, что и в исходной команде.
  - результат выполнения команды не должен убивать значения, которое потребуется в дальнейшем.
  - результат команды нельзя убивать до достижения *src.*
- Компенсирующий код потенциально в состоянии сделать некоторые из путей более медленными, так что перемещение кода повышает производительность программы, только тогда, когда оптимизированные пути выполняются более часто, чем неоптимизированные

3.6 Глобальное планирование кода

3.6.6 Перемещение кода *вниз* по пути управления

◊ Пусть команда из блока *src* перемещается вниз по пути управления в блок *dst*, не нарушая никаких зависимостей по данным.

- $src = Dom(dst)$ & $dst = Postdom(src)$:
  перемещенная команда выполняется только один раз
- $src \neq Dom(dst)$:
  существует хотя бы один путь, который достигает *dst,* минуя *src;* в этом случае будут выполняться дополнительные операции.
- $dst \neq Postdom(src)$:
  существует хотя бы один путь, который проходит через *src,* но не достигает *dst*; в этом случае в базовые блоки, которые образуют разрез, отделяющий *src* от *dst* необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях от *src* до *dst*).

  Замечание. Такой разрез размещается ниже блока *src*.

3.6 Глобальное планирование кода

3.6.7 Резюме по восходящему и нисходящему перемещениям кода

- ◊ Перемещение команд между эквивалентными по управлению базовыми блоками – простейшее и наиболее эффективное: не требуется ни дополнительных команд, ни компенсирующего кода.
- ◊ Если при перемещении кода вверх $src \neq Postdom(dst)$ (либо при перемещении кода вниз $src \neq Dom(dst)$), может потребоваться *выполнение дополнительных команд*. Поэтому такое перемещение кода выгодно в том случае, когда дополнительные операции могут быть выполнены бесплатно (с использованием только тех ресурсов, которые иначе простаивали бы), и когда выполняется путь, проходящий через src.
- ◊ Если при перемещении кода вверх $dst \neq Dom(src)$ (либо при перемещении кода вниз $dst \neq Postdom(src)$), требуется *компенсирующий код*. Пути с компенсирующим кодом могут оказаться замедленными, поэтому важно, чтобы оптимизированные пути выполнялись чаще, чем неоптимизированные.

3.4 Глобальное планирование кода

3.4.7 Резюме по восходящему и нисходящему перемещениям кода

◊ Случай, когда при перемещении кода вверх

$$src \neq Postdom(dst)\ \&\ dst \neq Dom(src)$$

(либо при перемещении кода вниз

$$src \neq Dom(dst)\ \&\ dst \neq Postdom(src)),$$

объединяет недостатки двух предыдущих случаев:

требуется компенсирующий код и может потребоваться выполнение дополнительных операций.
Поэтому в этом случае перемещение кода не производится

3.6 Глобальное планирование кода

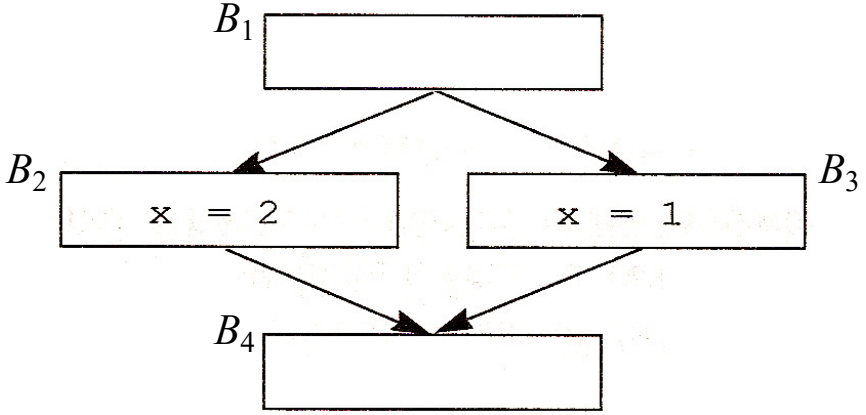
3.6.7 Резюме по восходящему и нисходящему перемещениям кода

◊ Различные варианты перемещения кода можно свести в следующую таблицу

↑ $src = Postdom(dst)$	$dst = Dom(src)$	Дублирование кода	Компенсирующий код
↓ $src = Dom(dst)$	$dst = Postdom(src)$		
+	+	–	–
–	+	+	–
+	–	–	+
–	–	+	+

3.6 Глобальное планирование кода

3.6.8 Поучительный пример



◊ В ГПУ на рисунке в блок B_1 можно перенести и присваивание **x = 2**, и присваивание **x = 1**, так как при переносе сохраняются все зависимости исходной программы.

◊ Однако после того, как одно из присваиваний будет перенесено в B_1, второе присваивание переносить будет нельзя: до перемещения переменная **x** на выходе из блока B_1 не является живой, а после перемещения она становится живой.

Если в некоторой точке программы переменная жива, то упреждающее определение этой переменной не может быть перемещено выше данной точки.

◊ Пример показывает, что перемещение кода может изменить отношения зависимостей по данным между командами.

◊ Таким образом, после каждого перемещения кода следует выполнять обновление зависимостей по данным

3.7 Алгоритмы глобального планирования

3.7.1 Вводные замечания

- ◊ Перемещение кода повышает эффективность некоторых путей за счет снижения эффективности остальных путей.
- ◊ Считается, что обычно 90% времени работы программы тратится на выполнение 10% ее кода.
- ◊ Цель как можно более ускорить часто выполнимые пути, за счет того, что более редкие пути замедлятся.

3.7 Алгоритмы глобального планирования

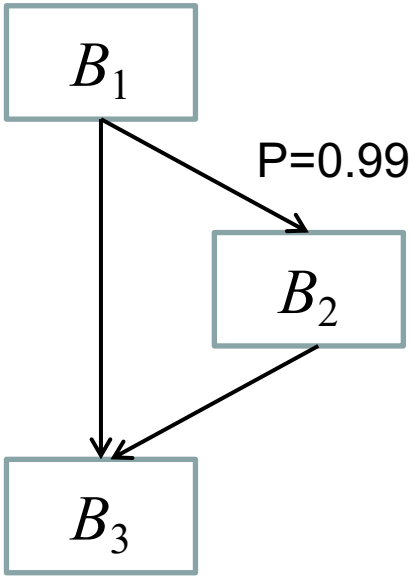
3.7.2 Оценка частоты выполнения путей

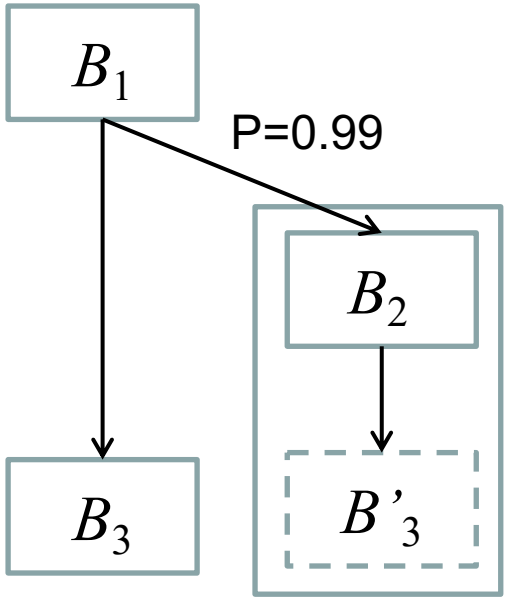
- Разумные соображения типа:
  - «команды во внутренних циклах выполняются гораздо чаще, чем код внешних циклов»
  - «при ветвлении чаще выбираются пути, ведущие назад, чем вперед»
  - «крайне редко отрабатываются ветвления, ведущие к выходу из программы или в подпрограммы обработки исключений»

  и т.п.
- Динамическое профилирование:
  - Программа *инструментируется*: в нее добавляется код для подсчета частоты выполнения ветвей
  - Выполняется тестовый запуск программы с «типичными входными данными».
  - Результаты, полученные при помощи этого метода, обычно оказываются достаточно точными.

    Результаты профилирования могут использоваться компилятором и для других оптимизаций.

“Treegions”





- больше команд и возможностей для их оптимального переупорядочения
- не требуется компенсационного кода при переносе из B3 в B2

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

- Рассмотрим планировщик, который поддерживает два вида перемещения кода:
  - перемещение команд вверх в блоки, эквивалентные с точки зрения управления;
  - упреждающее перемещение команд по одной из ветвей в предшествующий доминирующий блок.
- Алгоритм планирования использует области
  Напоминание:
  - область – это подмножество базовых блоков ГПУ, которые могут быть достигнуты через один входной базовый блок
  - существует три типа областей:
    - область-лист
    - область-тело
    - область-цикл
  - любая процедура может быть представлена как иерархия областей

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

- Порядок обхода графа потока:
  - Все обратные ребра удаляются – граф потока становится ациклическим
  - Все зависимости, идущие к заголовку области, удаляются из графа зависимостей – граф зависимостей становится ациклическим
  - Планирование кода ведется от внутренних областей к внешним: сначала планируется код в областях-листьях (базовых блоках), потом в областях-телах первого уровня, потом в областях-телах второго уровня и т.д.
  - При планировании области каждая вложенная подобласть рассматривается как черный ящик: команды не могут перемещаться в подобласть или из нее; однако команды могут перемещаться вокруг подобласти при условии удовлетворения ограничений зависимостей по данным и по управлению

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

- Порядок обхода графа потока (окончание):
  - Базовые блоки в каждой области посещаются в топологическом порядке. Такое упорядочение гарантирует, что базовый блок не будет планироваться до тех пор, пока не будут спланированы все команды, от которых он зависит.
  - Команды, которые должны быть спланированы в базовом блоке B, выводятся изо всех блоков, эквивалентных по управлению блоку B (включая сам базовый блок B), а также из непосредственных преемников блока B, над которыми он доминирует

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

◊ **Планирование внутри каждого базового блока** ведется с помощью алгоритма планирования списка, который поддерживает приоритетный список *CandInsts,* содержащий команды блоков-кандидатов, все предшественники которых уже спланированы

- В списке *CandInsts* используется функция приоритета, подобная функции приоритета для планирования списка, но с одним изменением – она обеспечивает выполнение команд из блоков-преемников блока только с упреждением: командам блоков, эквивалентных по управлению блоку B, назначается более высокий приоритет, чем командам блоков-преемников.
- План создается такт за тактом.
- Для каждого такта в порядке приоритета проверяется каждая команда n из списка *CandInsts* и включается в план, если позволяют ресурсы (функция $S(n)$).
- Затем обновляется список *CandInsts*
- Процесс повторяется до тех пор, пока не будут спланированы все команды из базового блока B.

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

- Формирование списка $CandInsts$ следующим образом:
  - Функция $ControlEquiv(B)$ строит множество блоков, эквивалентных блоку B по управлению
  - Функция $DominatedSucc,$ примененная к множеству базовых блоков, строит множество базовых блоков, являющихся преемниками как минимум одного блока множества, для которых все блоки множества являются доминирующими
- Построение множества $CandBlocks$:
  оно состоит из множества блоков, эквивалентных рассматриваемому блоку B по управлению, к которому добавлены доминируемые последователи
  В список $CandInsts$ включаются все уже спланированные команды из множества $CandBlocks$
- Формирование плана с помощью функции $S(n, B, t)$ – включение команды n из B блока в план на такт t.

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

- ◊ **Вход**: граф потока управления и описание машинных ресурсов.
- ◊ **Выход**: план S, отображающий каждую команду на базовый блок и интервал времени.
- ◊ **Метод**: выполнить следующую программу:

3.7 Алгоритмы глобального планирования

3.7.3 Алгоритм глобального планирования на основе областей

```
for (каждая область R в топологическом порядке, при котором
     внутренние области обрабатываются раньше внешних) {
     Вычисление зависимостей данных;
     for (каждый базовый блок B области R в приоритетном
     топологическом порядке ) {
          CandBlocks = ControlEquiv(B) ∪ DominatedSucc (ControlEquiv (B));
          CandInsts = готовые команды из CandBlocks;
          for (t = 0,1,..., пока не будут спланированы все
          команды из B) {
               for (каждая команда n из CandInsts в приоритетном порядке)
                    if (n не имеет конфликтов ресурсов в момент t) {
                              S(n, B, t);
                              Обновление имеющихся ресурсов;
                              Обновление зависимостей данных;
                    }
                    Обновление CandInsts;
          }
     }
}
```

3.7 Алгоритмы глобального планирования

3.7.4 Развертка циклов

◊ Планирование на основе областей ведется в рамках одной итерации цикла (удаление обратных ребер), так что граница итерации цикла является барьером для перемещения кода.

Простой, но эффективный метод упрощения этой проблемы состоит в частичной развертке цикла перед планированием кода.

◊ Развертывание создает в теле цикла большое количество команд, позволяя алгоритму глобального планирования достичь большей степени параллелизма.

3.7 Алгоритмы глобального планирования

3.7.4 Развертка циклов

◊ **Пример 1**. Цикл **`for`** вида:

```
for (i = 0; i < N; i++) {
      S(i) ;
}
```

можно переписать следующим образом:

```
for  (i  =  0;   i+4  < N;   i+=4)   {
      S(i) ;
      S(i+1);
      S(i+2);
      S(i+3) ;
}
for  (   ;   i  < N;)   {
      S (i) ;
}
```

3.7 Алгоритмы глобального планирования

3.7.4 Развертка циклов

◊ **Пример 2**. Цикл **`repeat`**

```
repeat
     S;
until C;
```

можно переписать следующим образом:

```
repeat {
     S;
     if (C) break;
     S;
     if (C) break;
     S;
     if (C) break;
     S;
} until C;
```

3.7 Алгоритмы глобального планирования

3.7.5 Планирование с добавлением компенсирующего кода

◊ Алгоритм 3.5.3 поддерживает только два вида перемещения кода:

- ◈ $B_1 = Dom(B_2)\ \&\ B_2 = Postdom(B_1)$
- ◈ $B_1 = Dom(B_2)\ \&\ B_2 \neq Postdom(B_1)$

◊ Однако может оказаться полезным перемещение, которое требует добавления компенсирующего кода, т.е. случай

$B_1 \neq Dom(B_2)\ \&\ B_2 = Postdom(B_1)$

Реализовать такое перемещение кода можно следующим образом:

- ◈ просматриваются все пары базовых блоков, выполняющиеся один за другим, и для каждой пары проверяется, нельзя ли переместить какую-либо команду между блоками пары для снижения общего времени выполнения этих блоков;
- ◈ когда требуемая пара блоков найдена, проверяется, не надо ли дублировать перемещаемую команду на других путях (компенсирующий код);
- ◈ перемещение выполняется только в том случае, когда оно дает выигрыш времени выполнения.

3.7 Алгоритмы глобального планирования

3.7.5 Планирование с добавлением компенсирующего кода

◊ Предложенное простое расширение алгоритма может оказаться эффективным для повышения производительности циклов:

В частности, оно может переместить команду из начала каждой итерации цикла, кроме первой, в конец предыдущей, а соответствующую команду из первой итерации вынести за пределы цикла. Такая оптимизация особенно полезна для небольших циклов, итерации которых состоят всего лишь из нескольких команд.

◊ Возможности этого метода ограничены, так как каждое решение о перемещении кода принимается локально и независимо от других.

Пример.

-O2

```
.L11:
        ldr     ip, [r3, r7]
        ldr     r4, [r3, r6]
        add     r3, r3, #4
        cmp     r3, r5
        mla     r2, r4, ip, r2
        mul     r1, r2, r1
        bne     .L11
```

2 cycles

5 cycles

До:
12 тактов

Source

```
int foo(int N) {
  int i;
  int a=0, b=1;
  for (i=0; i<N; i++)
  {
    a += x[i] * y[i];
    b *= a;
  }
  return a+b;
}
```

Selective Scheduling

```
Prologue
        ldr     r4, [r3, r7]
        mov     ip, #1
        ldr     r5, [r3, r8]
        mov     r2, r3
.L15:
Kernel
        mla     r2, r5, r4, r2
        cmp     r1, r6
        mov     r3, r1
        ldrne   r4, [r3, r7]
        addne   r1, r3, #4
        ldrne   r5, [r3, r8]
        mul     ip, r2, ip
        bne     .L15
```

После:
8 тактов

Ускорение
8%

Селективный планировщик выполняет конвейеризацию циклов

- «Скрывает» задержки у долгих команд, перенося их на предыдущую итерацию цикла
- Помогает использовать параллелизм на уровне команд
- Среднее ускорение 1-3%

3.7 Алгоритмы глобального планирования

3.7.6 Агрессивные алгоритмы перемещения кода

◊ Если целевой процессор статически имеет большие возможности по обеспечению параллелизма на уровне команд (например, *Intel Itanium*), может потребоваться более агрессивный алгоритм планирования кода.

◊ Агрессивный алгоритм можно получить путем усовершенствования алгоритма 3.5.3 в следующих направлениях:

- ⧫ Добавить новые базовые блоки в критические ребра графа потока управления
  (**Напоминание.** Критическим называется ребро, идущее от узла *Src* с более чем одним преемником к узлу *Dst* с более чем одним предшественником: $|Succ\ (Src)| > 1\ \&\ |Pred\ (Dst)| > 1$)

  Если в конце планирования кода добавленные блоки окажутся пустыми, они будут устранены.

  Добавление базовых блоков в критические ребра существенно упрощает структуру графа потока управления и облегчает его анализ.

3.7 Алгоритмы глобального планирования

3.7.6 Агрессивные алгоритмы перемещения кода

◊ Агрессивный алгоритм можно получить путем усовершенствования алгоритма 3.5.3 в следующих направлениях:

- ⟠ Одна из полезных эвристик состоит в перемещении команд из почти пустых базовых блоков, чтобы сделать их полностью пустыми и исключить из ГПУ.
- ⟠ Код, который будет выполняться в базовом блоке B, планируется раз и навсегда при посещении блока B. Такого простого подхода достаточно, так как алгоритм может перемещать операции только вверх в доминирующие блоки.

3.7 Алгоритмы глобального планирования

3.7.7 Взаимодействие с динамическим планировщиком

- ◊ Динамический планировщик – это, например, аппаратный планировщик OoO-процессора
- ◊ Динамический планировщик обладает тем преимуществом, что он может создавать новые планы в зависимости от условий времени выполнения, а не рассматривать заблаговременно все возможные планы.
- ◊ Если целевой процессор оснащен динамическим планировщиком (принадлежит классу OoO), основная функция статического планировщика состоит в обеспечении ранней выборки команд с большими задержками, чтобы динамический планировщик мог запланировать как можно более раннее выполнение этих команд. Это особенно эффективно в том случае, когда в целевом процессоре имеются команды предвыборки данных.
- ◊ В частности, предвыборка данных помогает предотвратить промахи кэша, представляющие собой класс непредсказуемых событий, которые могут привести к большим разбросам производительности программы.

3.7 Алгоритмы глобального планирования

3.7.7 Взаимодействие с динамическим планировщиком

- ◊ Если целевой процессор не оснащен динамическим планировщиком (не принадлежит классу OoO), статический планировщик должен быть консервативен и разделять все пары команд с зависимостями по данным минимальной задержкой (вставляя nop).
  Для достижения наилучшей производительности компилятор должен назначать большие задержки наиболее вероятным зависимостям и небольшие – маловероятным.
- ◊ Часто потеря производительности связана с неверным предсказанием ветвлений.