9. Планирование кода

9.1 Постановка задачи

9.1.1 Модель целевого процессора

- Каждое ядро современного процессора обеспечивает параллельное выполнение инструкций программы за счет
  - дублирования функциональных устройств (ФУ);
  - использования конвейерных ФУ;
  - возможности одновременной выдачи нескольких команд
- Рассмотрим модельный процессор (ядро), который за один такт может выполнить четыре операции:
  - одну загрузку,
  - одно сохранение,
  - одну арифметическую операцию
  - одну операцию перехода
- Для организации циклов имеется специальная операция перехода

```
BL R, L
```

которая

- уменьшает на единицу значение регистра **R** и
- если это значение $\neq 0$, передает управление на метку **L**.

9.1 Постановка задачи

9.1.1 Модель целевого процессора

- ◊ Операции с памятью могут выполняться в автоинкрементном режиме: если в команде после ссылки на регистр помещены символы **++**, то значение регистра автоматически увеличивается таким образом, чтобы на следующей операции с памятью регистр указывал на следующий адрес в памяти.
- ◊ Арифметические операции выполняются в конвейерном режиме. Они могут быть инициированы на любом такте, но их результаты становятся доступными два такта спустя.
- ◊ Операции с памятью выполняются три такта.
- ◊ Задержка всех прочих операций – один такт.

9.1 Постановка задачи

9.1.2 Пример

◊ Рассмотрим базовый блок, в котором вычисляется выражение

```
t = a * 2 * b * c * d
```

1	LD	R1	&a	
4	ADD	R1	R1	R1
5	LD	R2	&b	
8	MUL	R1	R1	R2
10	LD	R2	&c	
13	MUL	R1	R1	R2
15	LD	R2	&d	
18	MUL	R1	R1	R2
21	ST	&t	R1	

Без планирования кода
Результат на 24 такте

9.1 Постановка задачи

9.1.2 Пример

◊ Рассмотрим базовый блок, в котором вычисляется выражение

```
t = a * 2 * b * c * d
```

1	LD	R1	&a	
4	ADD	R1	R1	R1
5	LD	R2	&b	
8	MUL	R1	R1	R2
10	LD	R2	&c	
13	MUL	R1	R1	R2
15	LD	R2	&d	
18	MUL	R1	R1	R2
21	ST	&t	R1	

Без планирования кода
Результат на 24 такте

1	LD	R1	&a	
2	LD	R2	&b	
3	LD	R3	&c	
4	ADD	R1	R1	R1
5	MUL	R1	R1	R2
6	LD	R2	&d	
7	MUL	R1	R1	R3
9	MUL	R1	R1	R2
11	ST	&t	R1	

После планирования кода
Результат на 14 такте

9.1 Постановка задачи

9.1.3 Цель планирования кода

- Цель планирования кода – выбрать такую последовательность команд, которая, **не меняя семантики программы**, обеспечит оптимальное использование особенностей архитектуры целевого процессора и, прежде всего, правильное использование возможностей параллельного выполнения команд, реализованных в его аппаратуре
- Ответ на вопрос:
  - насколько быстро может выполняться программа на процессоре с параллелизмом на уровне команд?

  зависит от следующих факторов:
  - Доступный параллелизм процессора.
  - Потенциальный параллелизм программы.
  - Возможность выделить параллелизм в исходной последовательной программе.
  - Наша способность спланировать наилучшее параллельное выполнение при заданных ограничениях планирования.

9.1 Постановка задачи

9.1.4 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 машины.

9.2 Анализ зависимостей по данным

9.2.1 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 по данным:

- *Истинная зависимость*: **чтение после записи**.
  Если команда C_1 записывает значение в некоторую ячейку памяти (или на регистр), 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.

9.2 Анализ зависимостей по данным

9.2.1 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 по данным:

⟐ *Истинная зависимость*: **чтение после записи**.
Если команда C_1 записывает значение в некоторую ячейку памяти, а команда C_2 записывает в эту ячейку новое значение, то команды C_1 и C_2 зависимы.

Неустранимой является только истинная зависимость.
Два других вида зависимостей (их называют *зависимостями, связанными с хранением*) могут быть устранены: нужно использовать в командах C_1 и C_2 разные ячейки памяти.

⟐ *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.

9.2 Анализ зависимостей по данным

9.2.2 Требование консервативности анализа

◊ **Консервативность**. Компилятор обязан считать, что две команды могут обращаться к одним и тем же ячейкам, если он не может доказать обратное.

◊ **Пример**: Рассмотрим код

```
1. a ← 1
2. *p ← 2
3. b ← a
```

Сразу можно обнаружить истинную зависимость между командами 1 и 3.

Больше зависимостей как будто нет, но это только в том случае, если *компилятор может доказать,* что указатель **p** не может указывать на **a**.

В противном случае, 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.

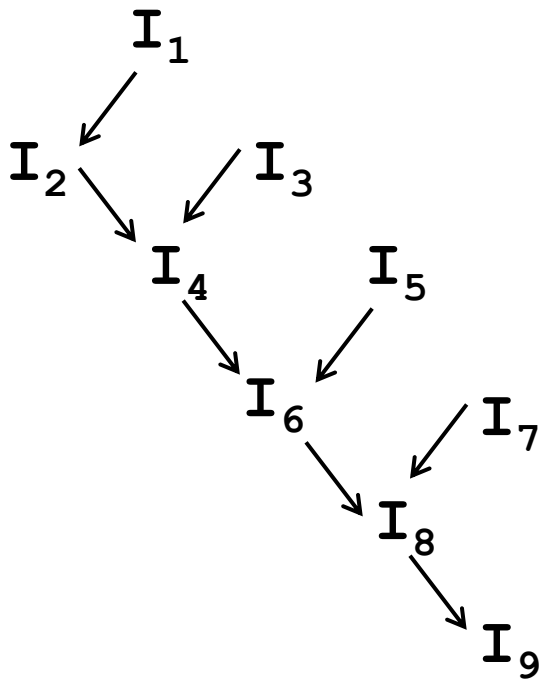
9.2 Анализ зависимостей по данным

9.2.3 Граф зависимостей (по данным)

◊ В базовом блоке *B граф зависимостей D* – ориентированный граф, вершинами которого являются команды блока , а дуги соединяют вершины n_1 и n_2, если n_2 использует результат n_1.

◊ На рисунке изображен граф зависимостей для примера 9.1.2

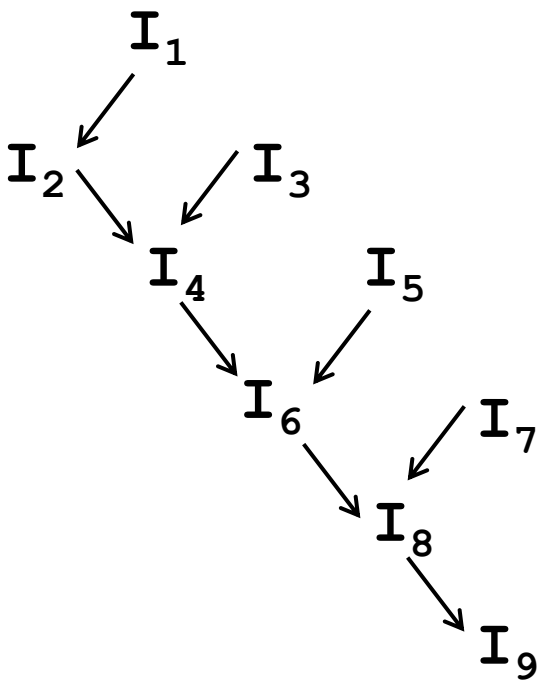
I_1	LD	R1	&a	
I_2	ADD	R1	R1	R1
I_3	LD	R2	&b	
I_4	MUL	R1	R1	R2
I_5	LD	R2	&c	
I_6	MUL	R1	R1	R2
I_7	LD	R2	&d	
I_8	MUL	R1	R1	R2
I_9	ST	&t	R1	



9.2 Анализ зависимостей по данным

9.2.3 Граф зависимостей (по данным)

- ◊ Учитываются только истинные зависимости.
- ◊ Вершина графа D, у которой нет последователей, называется корнем графа D.
- ◊ В рассматриваемом примере граф D – дерево. В общем случае граф D – лес, и у него несколько корней.
- ◊ *Расписание* S сопоставляет каждой операции o номер такта $i = S(o)$, в который она может начать выполняться.
- ◊ i-я команда содержит все операции $\{o \mid S(o) = i\}$



9.3 Локальное планирование

9.3.1 Расписание (порядок выдачи) команд

◊ Расписание $S(o)$ должно удовлетворять следующим ограничениям:

1. Для любой операции o $S(o) \geq 1$, т.е. операции не могут быть выданы до начала выполнения программы. Кроме того, расписание должно содержать по крайней мере одну операцию o, у которой $S(o) = 1$.
2. Если o_1 и o_2 – концы дуги графа зависимостей, то для обеспечения корректности необходимо, чтобы $S(o_1) + delay(o_1) \leq S(o_2)$, т.е. операция не может быть выдана до того, как будут определены ее операнды.
3. Каждая команда может содержать не больше однотипных операций, чем это допускается системой команд целевой машины (гарантия выполнимости).

9.3 Локальное планирование

9.3.3 Расписание (порядок выдачи) команд

◊ По определению продолжительность расписания $S(o)$ фрагмента программы P – это номер такта, на котором завершится последняя операция. Продолжительность $L(S)$ расписания S можно вычислить по формуле

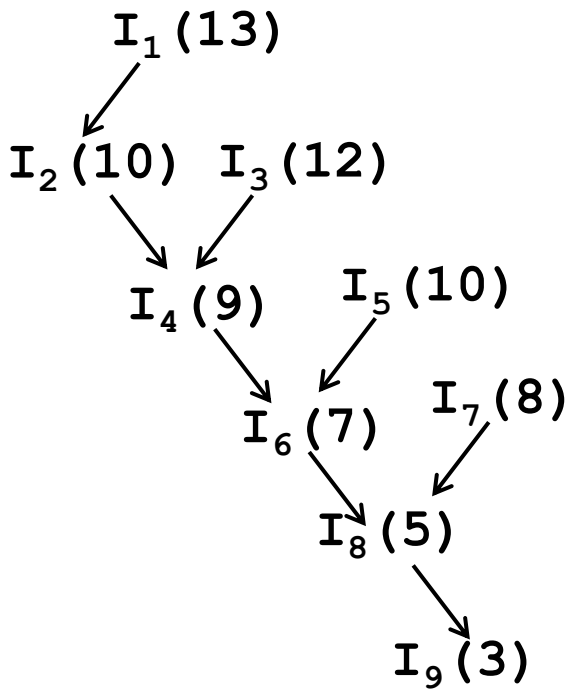
$$L(S) = \max_{o \in P}(S(o) + delay(o))$$

◊ *Оптимальным по времени расписанием* для фрагмента программы P называется расписание $S_{opt}(P)$, которое удовлетворяет условию: для любого расписания $S(P) \neq S_{opt}(P)$ $L(S(P)) \geq L(S_{opt}(P))$

9.3 Локальное планирование

9.3.2 Простой алгоритм планирования

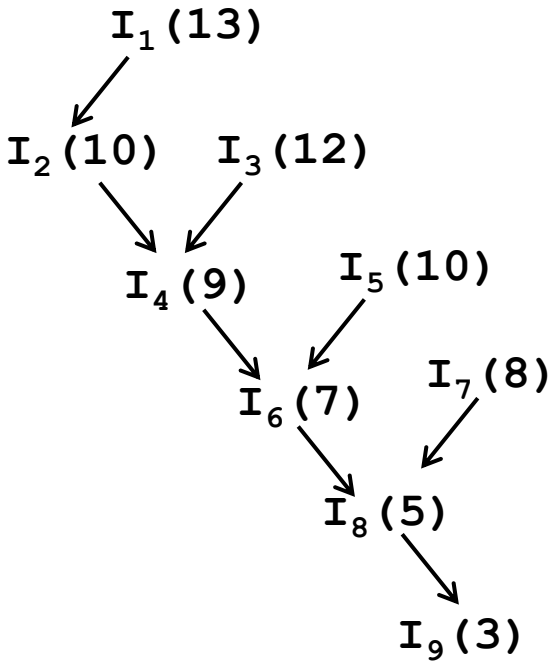
- ◊ Вершина графа D, у которой нет последователей, называется корнем графа D.
- ◊ В рассматриваемом примере граф D – дерево. В общем случае граф D – лес, и у него несколько корней.
- ◊ *Расписание* S сопоставляет каждой операции o номер такта $i = S(o)$, в который она может начать выполняться.
- ◊ i-я команда содержит все операции $\{o \mid S(o) = i\}$
- ◊ Путь $\mathtt{I_1 I_2 I_4 I_6 I_8 I_9}$ – самый длинный (*критический*); он определяет полное время выполнения рассматриваемого фрагмента



Числа в скобках равны продолжительности вычислений от соответствующей вершины до окончания вычислений они помогают составить расписание

9.3 Локальное планирование

9.3.2 Простой алгоритм планирования



◊ Расписание для фрагмента. На первое место в нем претендуют вершины I_1, I_3, I_5, I_7, так как у них нет предшественников и их операнды готовы; но I_1 находится на критическом пути, так что именно эту операцию нужно запланировать первой.

◊ Продолжая рассуждать подобным образом, получим расписание $I_1I_3I_5I_2I_4I_7I_6I_8I_9$.

◊ У полученного расписания существенный недостаток – при его составлении не учитывались зависимости по данным: I_3 и I_5 определяют **R2**, а I_4 использует значение **R2**, установленное I_3. Так что I_5 нельзя помещать между I_3 и I_4.

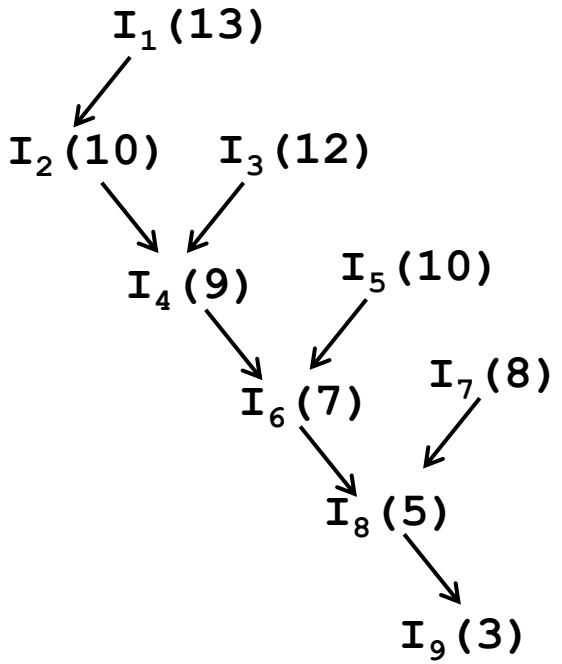
I_1	LD	R1	&a	
I_3	LD	R2	&b	
I_5	LD	R2	&c	
I_2	ADD	R1	R1	R1
I_4	MUL	R1	R1	R2
I_7	LD	R2	&d	
I_6	MUL	R1	R1	R2
I_8	MUL	R1	R1	R2
I_9	ST	&t	R1	

9.3 Локальное планирование

9.3.2 Простой алгоритм планирования

◊ Но между I_3 и I_5 – антизависимость, и все можно решить, «переименовав» регистр **R2** в I_5, т.е. заменить **LD R2 &c** на **LD R3 &c**.

◊ Если ввести обозначение антизависимости между I_3 и I_5 как $I_5 \rightarrow I_3$, то в рассматриваемом примере есть еще три антизависимости: $I_5 \rightarrow I_4$, $I_7 \rightarrow I_5$, и $I_7 \rightarrow I_6$. Их тоже можно разрешить, использовав дополнительные регистры, но тут возникают сложности в связи с нехваткой регистров.



I_1	LD	R1	&a	
I_3	LD	R2	&b	
I_5	LD	R2	&c	
I_2	ADD	R1	R1	R1
I_4	MUL	R1	R1	R2
I_7	LD	R2	&d	
I_6	MUL	R1	R1	R2
I_8	MUL	R1	R1	R2
I_9	ST	&t	R1	

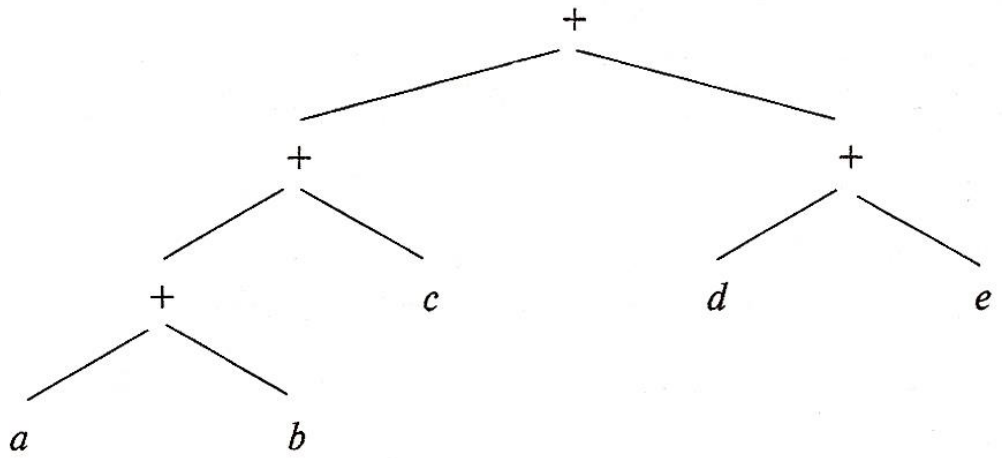
9.3 Локальное планирование

9.3.3 Зависимости по данным и распределение регистров

◊ **Пример**. Рассмотрим код, вычисляющий выражение

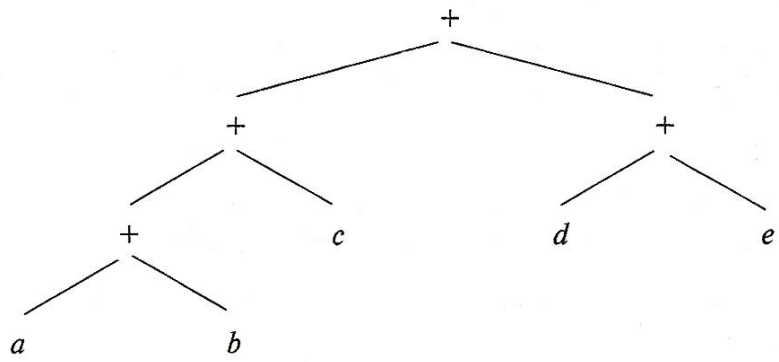
```
(a + b)   +  c +   (d + e)
```

(скобки в выражении расставлены в соответствии с деревом, показанном на рисунке)



9.3 Локальное планирование

9.3.3 Зависимости по данным и распределение регистров




```
1. LD   R1,&a          //R1 = a
2. LD   R2,&b          //R2 = b
3. ADD  R1,R1,R2       //R1 = R1+R2
4. LD   R2,&c          //R2 = c
5. ADD  R1,R1,R2       //R1 = R1+R2
6. LD   R2,&d          //R2 = d
7. LD   R3,&e          //R3 = e
8. ADD  R2,R2,R3       //R2 = R2+R3
9. ADD  R1,R1,R2       //R1 = R1+R2
```

◊ Экономное распределение регистров приведет к последовательному коду, представленному вверху справа. Этот код использует всего три регистра, но параллельно в нем можно выполнить только загрузки регистров в строках 1 и 2 и загрузки регистров в строках 6 и 7.
Таким образом, для вычисления выражения с максимальным использованием параллельности потребуется 7 шагов.

9.3 Локальное планирование

9.3.3 Зависимости по данным и распределение регистров

```
1. LD  R1,&a        //R1 = a
2. LD  R2,&b        //R2 = b
3. ADD R1,R1,R2     //R1 = R1+R2
4. LD  R2,&c        //R2 = c
5. ADD R1,R1,R2     //R1 = R1+R2
6. LD  R2,&d        //R2 = d
7. LD  R3,&e        //R3 = e
8. ADD R2,R2,R3     //R2 = R2+R3
9. ADD R1,R1,R2     //R1 = R1+R2
```

◊ Если использовать различные регистры для каждой промежуточной суммы, то выражение можно вычислить за 4 шага, что равно высоте дерева выражения.
Но при этом потребуется не 3, а 8 регистров.

```
1. LD  R1,&a |
2. LD  R2,&b |
3. LD  R3,&c |          шаг 1
4. LD  R4,&d |
5. LD  R5,&e |
6. ADD R6,R1,R2 ||      шаг 2
7. ADD R7,R4,R5 ||
8. ADD R8,R6,R3 |||     шаг 3
9. ADD R8,R8,R7 ||||    шаг 4
```

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

◊ Алгоритм включает следующие четыре шага

1. *Переименование значений, чтобы избежать антизависимостей*.
   Значения переименовываются таким образом, чтобы каждое значение имело уникальное имя. Это нужно для того, чтобы найти и те расписания, которые были бы исключены из-за антизависимостей.
2. *Построение графа зависимостей D*.
   Базовый блок обходится снизу вверх. Для каждой операции строится вершина графа, представляющая значение, вычисленное этой операцией. Построенная вершина соединяется дугами с вершинами, использующими это значение.

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

◊ Алгоритм включает следующие четыре шага

3. *Присваивание приоритета каждой операции.*
   Планировщик использует эти приоритеты при выборе операций. Каждая вершина снабжается несколькими оценками, используемыми планировщиком для определения очередной операции, которая должна быть запланирована, и для выбора операций, когда основная оценка у них одинакова. Известно несколько различных схем назначения приоритетов. Классическая схема использует в качестве основного приоритета продолжительность вычисления узла с учетом задержек.
4. *Итеративный процесс выбора вершин и планирования их*.
   Алгоритм устанавливает счетчик тактов в 1 и пытается запланировать как можно больше операций. Потом увеличивает счетчик тактов и переходит к планированию следующего такта. И так далее пока все операции не будут запланированы.

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

- ◊ Первые два шага (переименование и построение графа **D**) не нуждаются в комментариях.
- ◊ Типичное вычисление приоритетов (шаг 3) состоит в обходе графа зависимостей **D** и вычислении некоторых метрик.
- ◊ Основная часть алгоритма – это алгоритм планирования (АП), псевдокод которого (в предположении, что у целевого процессора всего одно ФУ) представлен на следующем слайде. АП выполняет абстрактное моделирование (симуляцию) процесса выполнения блока, уделяя основное внимание ограничениям по времени, налагаемыми дугами графа **D**.

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

◊

```
Cycle ← 1
Ready ← leaves of D
Active ← ∅
while (Ready ∪ Active ≠ ∅){
    for each op ∈ Active
        if (S(op) + delay(op) < Cycle)then
            remove op from Active
            for each successor s of op in D
                if s is ready
                    then add s to Ready
     if Ready ≠ ∅ then
            remove an op from Ready
            S(op) ← Cycle
            add op to Active
            Cycle ← Cycle + 1
```

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

- ◊ Значением переменной **Cycle** (такт) является номер текущего такта.
- ◊ Список **Ready** содержит все операции, которые можно выполнить на текущем такте. Сначала **Ready** содержит все листья графа **D**, так как они не зависят от других операций блока.
- ◊ Список **Active** содержит все операции, которые начали выполняться на предыдущих тактах, но пока не завершились. Когда номер текущего такта меняется, из исключаются все операции, завершающиеся на новом такте
- ◊ Качество полученного расписания зависит прежде всего от механизма, используемого для выборки операции из очереди **Ready**.

  В простейшем случае, когда **Ready** содержит не более одного элемента на каждой итерации, алгоритм сгенерирует оптимальное расписание. В самом деле, на каждом такте либо есть единственная готовая операция, либо нет ни одной готовой операции, и у алгоритма не возникает вопросов, что запланировать.

9.3 Локальное планирование

9.3.4 Алгоритм планирования с помощью списков

- ◊ Когда алгоритм должен выбирать из нескольких готовых операций, все зависит от качества алгоритма выбора. Алгоритм выбирает операцию с наивысшим приоритетом. Если приоритеты равны требуется рассмотреть дополнительные критерии.
- ◊ Операции обмена с памятью часто имеют неопределенные и непостоянные задержки: этому способствуют промахи кэшей и прочие особенности работы с памятью. Фактическая задержка при этом может меняться от 0 до нескольких тысяч тактов. Если планировщик исходит из наихудшей возможности, он рискует получить достаточно длительные простои процессора, если из наилучшей, он спровоцирует дополнительные промахи кэша.
- ◊ Наиболее привлекательным представляется алгоритм *сбалансированного планирования*, когда задержка оценивается в процессе планирования в зависимости от контекста (а также, возможно, характеристик целевого процессора).

9.4 Глобальное планирование

9.4.1 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, что:
  - ◈ оптимизированная программа выполняет все команды исходной программы;
  - ◈ при выполнении некоторых команд исходной программы с упреждением не возникает никаких побочных действий.

9.4 Глобальное планирование кода
9.4.2 Зависимости по управлению

◊ Команда **`s1`** *зависит по управлению* от команды **`s2`**, если выполнение команды **`s2`** определяет, будет ли выполняться команда **`s1`**.

Простые примеры

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

9.4 Глобальное планирование кода

9.4.2 Зависимости по управлению

◊ **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

9.4 Глобальное планирование кода

9.4.2 Зависимости по управлению

◊ **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

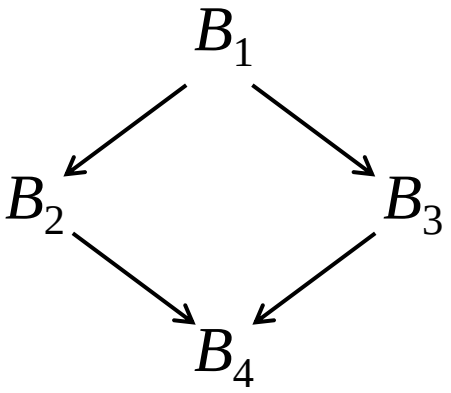
9.4 Глобальное планирование кода

9.4.3 Эквивалентность по управлению

◊ **Определение**. Если базовый блок B является доминатором базового блока B', а блок B' - постдоминатором блока B, то блоки B и B' называются *эквивалентными по управлению*, что означает, что блок B выполняется тогда и только тогда, когда выполняется блок B'.

◊ Возможны следующие отношения между блоками:

- ◈ Блоки B_1 и B_4 эквивалентны по управлению: $B_1 = Dom(B_4)$ & $B_4 = Postdom(B_1)$
- ◈ Блок B_1 доминирует над блоком B_2, а блок B_2 не постдоминирует над блоком B_1.
- ◈ Блок B_2 не доминирует над блоком B_4, а блок B_4 постдоминирует над блоком B_2.
- ◈ Блоки B_2 и B_3 не связаны ни отношением доминирования, ни отношением постдоминирования.



9.4 Глобальное планирование кода

9.4.4 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.

- если *Dst* **доминирует над** *Src,* перемещенная команда выполняется только один раз
- если *Dst* **не доминирует над** *Src,* существует один или несколько путей, которые достигают *Src,* но не проходят через *Dst;* в этом случае в базовые блоки, которые образуют разрез, отделяющий входной блок от *Src* необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях, достигающих *Src*).

9.4 Глобальное планирование кода

9.4.4 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.

- если *Src* **не постдоминирует над** *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 лишние команды не должны приводить к нежелательным побочным эффектам, иначе перемещение кода **некорректно**

9.4 Глобальное планирование кода

9.4.4 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 компенсирующая 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 если оптимизированные пути выполняются более часто, чем неоптимизированные

9.4 Глобальное планирование кода

9.4.5 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.

- если *Dst* **постдоминирует над** *Src*,
  то перемещенная команда выполняется только один раз
- если *Src* **не доминирует над** *Dst*,
  то существует один или несколько путей, которые достигают *Dst*, минуя *Src*; в этом случае будут выполняться дополнительные операции.

9.4 Глобальное планирование кода

9.4.5 Перемещение кода вниз по пути управления

◊ Если перемещение кода вниз применяется к операциям записи, может возникнуть побочный эффект – убивается старое значение. Обойти это можно:

- либо создав дубликаты базовых блоков на пути от *Src* к *Dst* и разместив команды записи в новых копиях блоков;
- либо, при возможности, используя предикатные команды таким образом, чтобы они были защищены тем же предикатом, что и блок *Src;*
  для этого предикатная команда должна размещаться в блоке, над которым доминирует вычисление предиката, так как иначе предикат будет недоступен.

9.4 Глобальное планирование кода

9.4.5 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.

⟠ Если *Dst* **не постдоминирует над** *Src*, существует один или несколько путей, которые проходят через *Src,* но не достигают *Dst;* в этом случае в базовые блоки, которые образуют разрез, отделяющий *Src* от выходного блока необходимо вставить *компенсирующий код*: копии перемещаемой команды (чтобы она выполнялась на всех путях от *Src* до выхода).

Замечание. Разрез, отделяющий блок *Src* от выхода, размещается ниже блока *Src*.

9.4 Глобальное планирование кода

9.4.6 Резюме по восходящему и нисходящему перемещению 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 $Src\ !postdom\ Dst$ (либо при перемещении кода вниз $Src\ !dom\ Dst$), 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.

9.4 Глобальное планирование кода

9.4.6 Резюме по восходящему и нисходящему перемещению кода

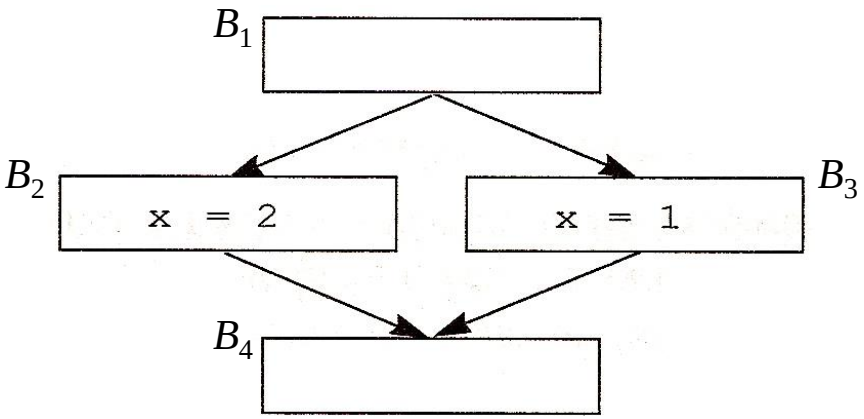
- ◊ Если при перемещении кода вверх *Dst !dom Src*
  (либо при перемещении кода вниз *Dst !postdom Src*),
  требуется *компенсирующий код*.
  Пути с компенсирующим кодом могут оказаться замедленными, поэтому важно, чтобы оптимизированные пути выполнялись чаще, чем неоптимизированные.
- ◊ Если при перемещении кода вверх
  Src !postdom Dst & Dst !dom Src
  (либо при перемещении кода вниз
  Src !dom Dst & Dst !postdom Src),
  требуется *компенсирующий код* и может потребоваться *выполнение дополнительных команд*

9.4 Глобальное планирование кода

9.4.7 Замечания

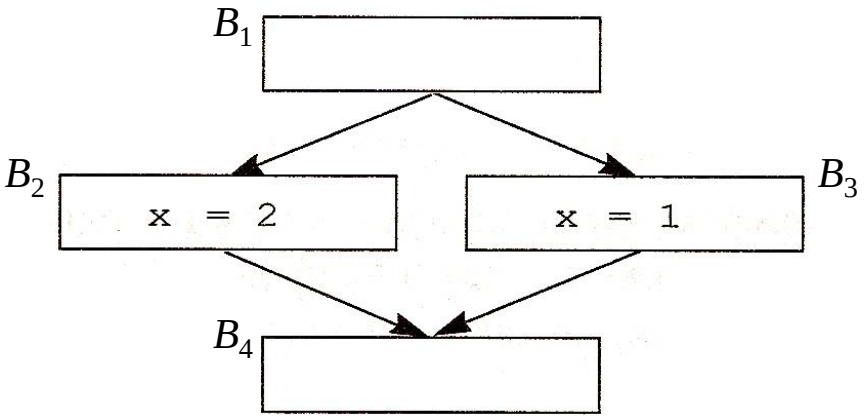
- В примере на рисунке в блок B_1 может быть перенесено и присваивание **x = 2**, и присваивание **x = 1**, так как при таком преобразовании сохраняются все зависимости исходной программы.
- Однако после того, как одно из присваиваний будет перенесено в B_1, второе переносить будет нельзя, так как до перемещения переменная **x** не является живой на выходе из блока B_1, а после перемещения становится живой.

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



9.4 Глобальное планирование кода

9.4.7 Замечания



- ◊ Пример также показывает, что перемещение кода может изменить отношения зависимостей по данным между командами.
- ◊ Таким образом, после каждого перемещения кода следует выполнять обновление зависимостей по данным

9.4 Глобальное планирование кода

9.4.8 Алгоритм глобального планирования на основе областей

- Рассмотрим планировщик, который поддерживает две простейшие разновидности перемещения кода:
  - перемещение команд вверх в блоки, эквивалентные с точки зрения управления;
  - упреждающее перемещение команд по одной из ветвей в предшествующий доминирующий блок (тоже вверх).
- Алгоритм планирования использует области (регионы)
  - 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 (имеется в виду естественный цикл с единственной обратной дугой)
  - любая процедура может быть представлена как иерархия областей

9.4 Глобальное планирование кода

9.4.8 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

9.4 Глобальное планирование кода

9.4.8 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

9.4 Глобальное планирование кода

9.4.8 Алгоритм глобального планирования на основе областей

- **Планирование внутри каждого базового блока** ведется с помощью алгоритма планирования списком, который поддерживает $CandInsts$ – приоритетный список команд-кандидатов, содержащий команды блоков-кандидатов, все предшественники которых уже спланированы
  - В списке $CandInsts$ используется функция приоритета, подобная функции приоритета для планирования списком, но с одним важным изменением (оно 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.

9.4 Глобальное планирование кода

9.4.8 Алгоритм глобального планирования на основе областей

- **Планирование внутри каждого базового блока** ведется с помощью алгоритма планирования списком, который поддерживает $CandInsts$ – приоритетный список команд-кандидатов, содержащий команды блоков-кандидатов, все предшественники которых уже спланированы
  - 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.

9.4 Глобальное планирование кода

9.4.8 Алгоритм глобального планирования на основе областей

- Список $CandInsts$ формируется 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
  - Строится множество $CandBlocks$: оно состоит из множества блоков, эквивалентных рассматриваемому блоку B по управлению, к которому добавлены доминируемые последователи
  - В список 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.

9.4 Глобальное планирование кода

9.4.9 Развертка циклов

◊ Планирование на основе областей ведется в рамках одной итерации цикла, 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.

◊ Цикл **`for`** вида:

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

можно переписать следующим образом:

9.4 Глобальное планирование кода

9.4.9 Развертка циклов

◊ 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) ;
}
```

9.4 Глобальное планирование кода

9.4.9 Развертка циклов

◊ 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;
```

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

9.4 Глобальное планирование кода

9.4.10 Взаимодействие с динамическим планировщиком

- ◊ Динамический планировщик – это, как правило, аппаратный планировщик *OoO*-процессора
- ◊ Динамический планировщик обладает тем преимуществом, что он может создавать новые планы в зависимости от условий времени выполнения, и 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.