6. Оптимизация циклов (часть I)

6.1. Выделение естественных циклов

6.1.1 Классификация дуг ГПУ

◊ Дуги ГПУ, являющиеся дугами и его остовного дерева, называются *остовными*.

◊ Дуги ГПУ, не являющиеся дугами его остовного дерева, но имеющие такое же направление, что и остовные, называются *прямыми*.

◊ Дуги ГПУ, направленные противоположно остовным, называются *обратно направленными*.

◊ Обратно направленная дуга ГПУ $\langle B_i, B_k \rangle$ называется *обратной*, если $B_k = Dom(B_i)$

◊ Остальные дуги ГПУ называются *поперечными*. Поперечные дуги соединяют различные поддеревья остовного дерева и в программах нормальных программистов не встречаются, так как нормальные программисты не злоупотребляют применением `goto`.

6.1. Выделение естественных циклов

6.1.2 Определение естественного цикла

◊ **Определение**. *Естественным циклом* называется цикл со следующими свойствами:

- ♦ В графе потока управления цикл имеет единственную входную вершину, называемую его *заголовком*,
- ♦ Существует обратная дуга, ведущая в заголовок цикла.

◊ **Определение**. *Естественный цикл обратной дуги* $\langle B_i, B_k \rangle$ составляют узел B_k (*заголовок цикла*) и все узлы ГПУ, из которых можно достичь узла B_i, не проходя через узел B_k.(эти узлы составляют *тело цикла*).

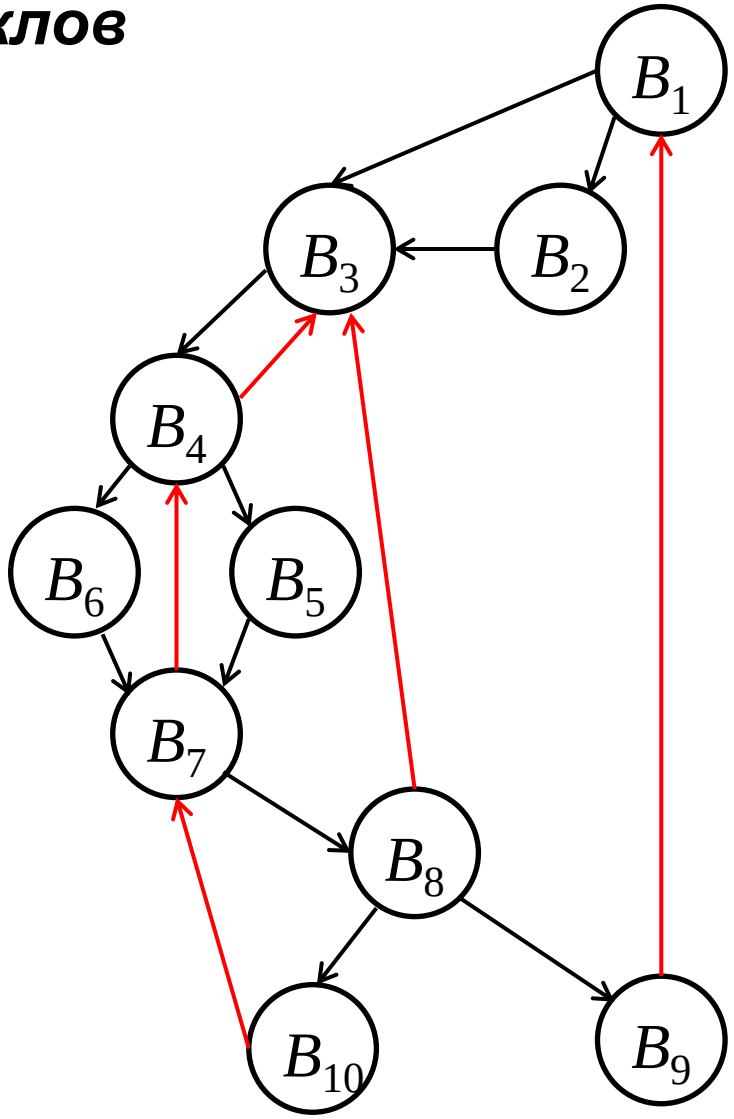
6.1. Выделение естественных циклов

6.1.3 Примеры естественных циклов

◊ На рисунке справа – пять обратных дуг:
$\langle B_{10}, B_7\rangle$, $\langle B_7, B_4\rangle$, $\langle B_4, B_3\rangle$,
$\langle B_8, B_3\rangle$, $\langle B_9, B_1\rangle$

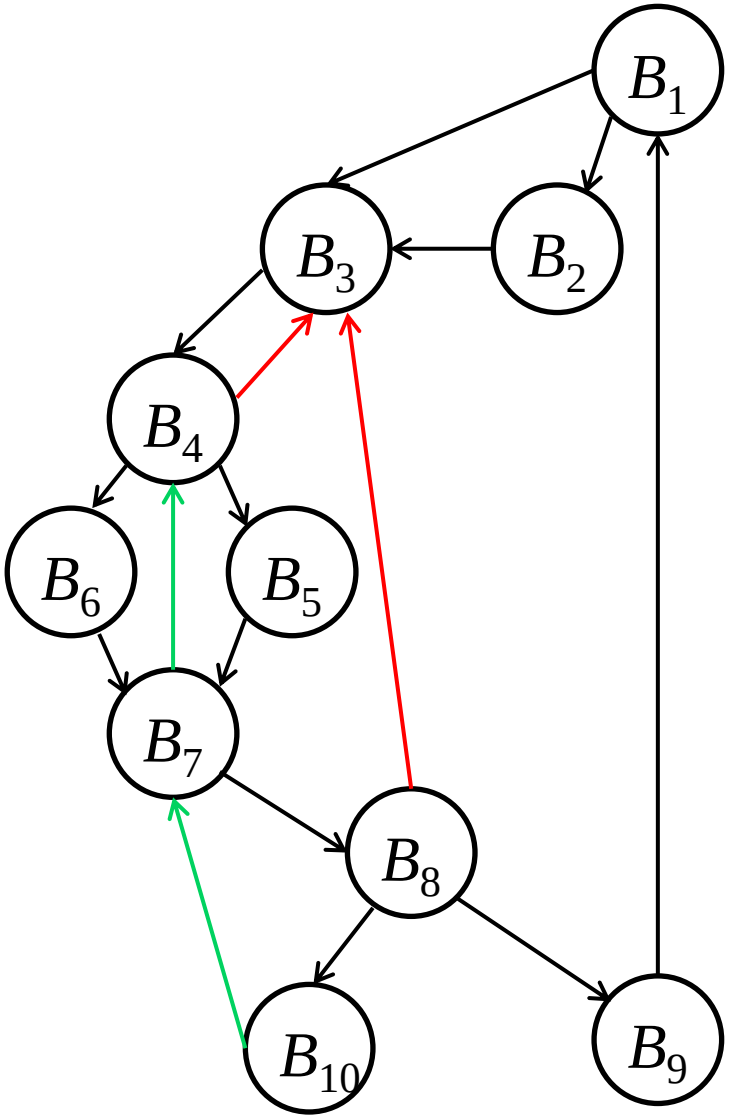
◊ Обратной дуге $\langle B_{10}, B_7\rangle$ соответствует естественный цикл $\{B_7, B_8, B_{10}\}$, так как из вершин B_8 и B_{10} можно достичь вершины B_{10}, не проходя через B_7.

◊ Обратной дуге $\langle B_7, B_4\rangle$ соответствует естественный цикл
$\{B_4, B_5, B_6, B_7, B_8, B_{10}\}$
Этот цикл включает в себя цикл дуги $\langle B_{10}, B_7\rangle$, который является вложенным циклом



6.1. Выделение естественных циклов

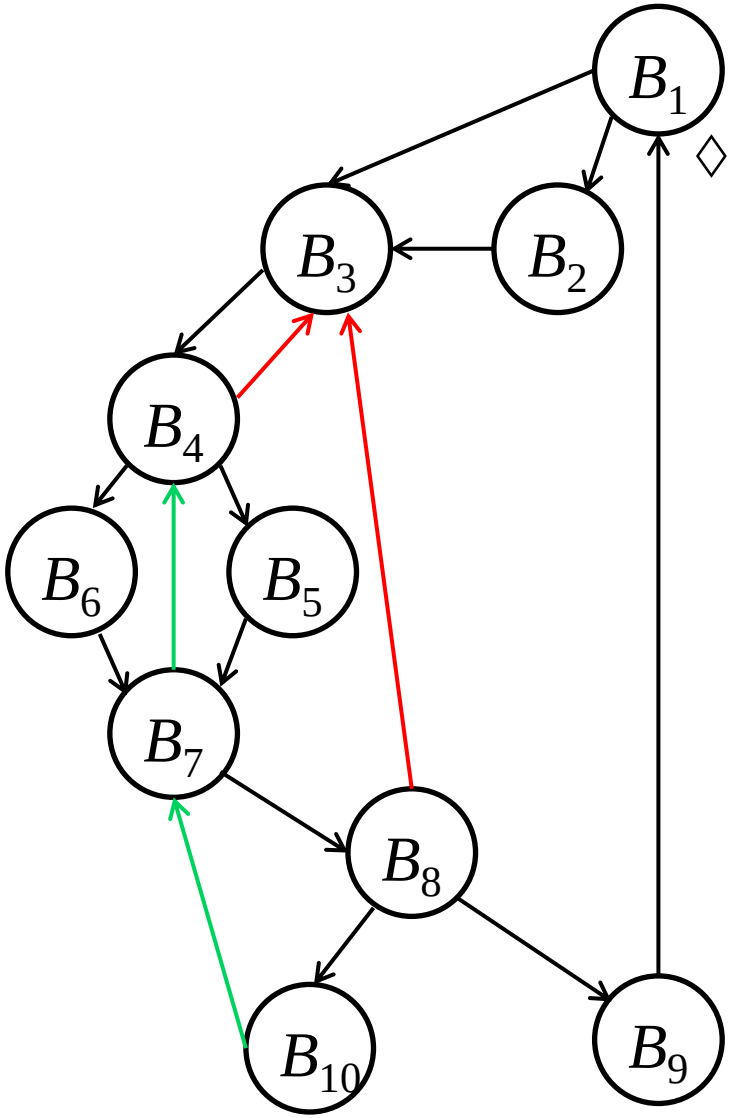
6.1.3 Примеры естественных циклов



◊ У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$

6.1. Выделение естественных циклов

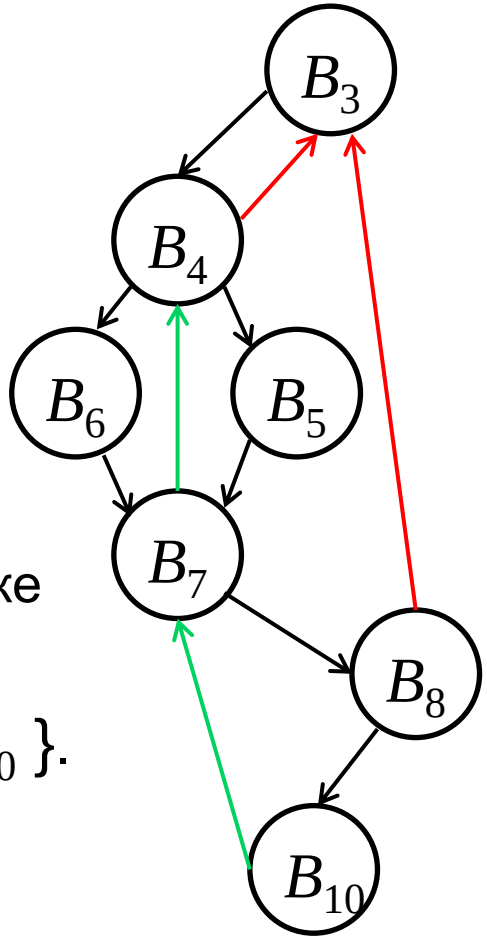
6.1.3 Примеры естественных циклов



◊

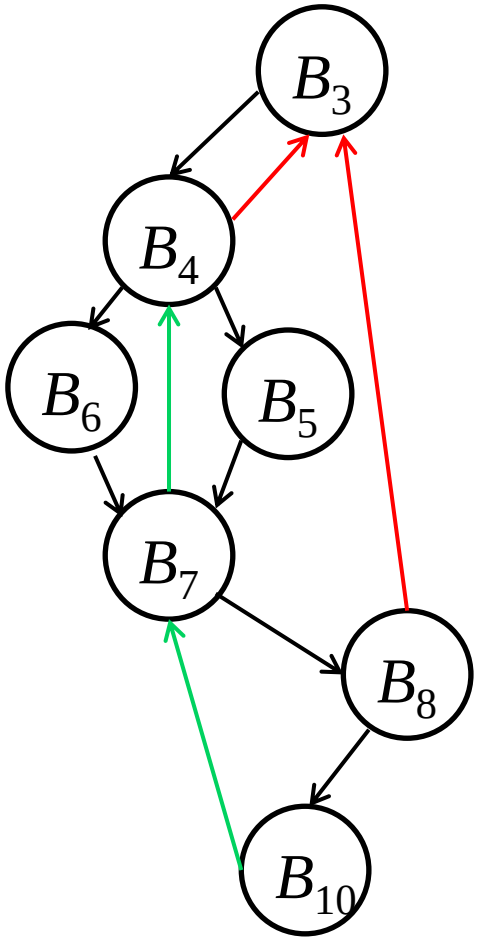
У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один.

В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$

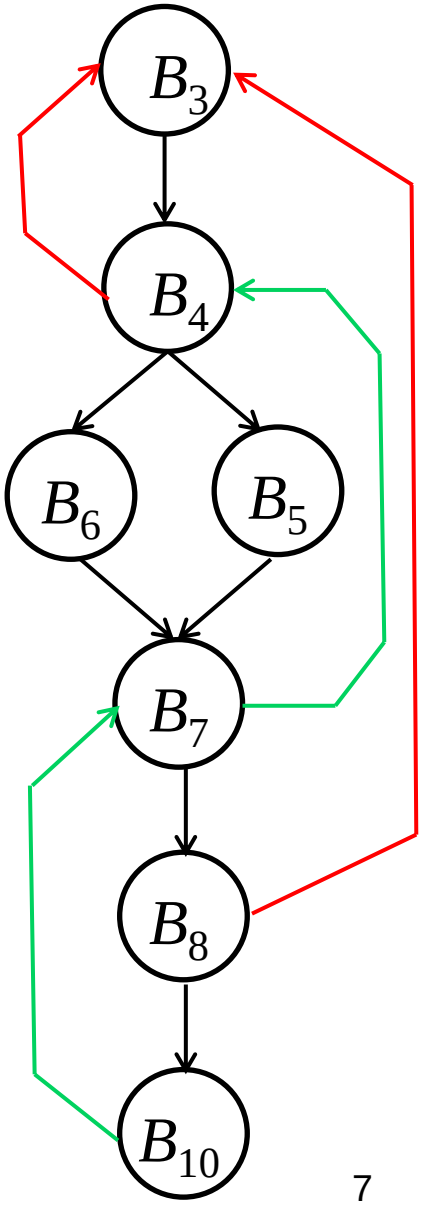


6.1. Выделение естественных циклов

6.1.3 Примеры естественных циклов



- У естественных циклов обратных дуг $\langle B_4, B_3 \rangle$ и $\langle B_8, B_3 \rangle$ один и тот же заголовок B_3 и одно и то же множество вершин $\{B_3, B_4, B_5, B_6, B_7, B_8, B_{10}\}$. Эти два цикла можно объединить в один. В объединенный цикл вложены циклы обратных дуг $\langle B_{10}, B_7 \rangle$ и $\langle B_7, B_4 \rangle$
- При этом изображение ГПУ справа более понятно, чем изображение слева.



6.1. Выделение естественных циклов

6.1.4 Алгоритм построения естественного цикла по обратной дуге

Вход: ГПУ $G = \langle N, E \rangle$ с входным узлом $Entry$.

Обратное ребро $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся естественным циклом.

Метод:

(1) начальное значение C – множество $\{n, d\}$.

(2) узел d помечается как «посещенный».

(3) начиная с узла n выполняется поиск в глубину на обратном ГПУ (направления дуг заменены на противоположные).

(4) все узлы, посещенные на шаге (3), добавляются в C.

6.1. Выделение естественных циклов

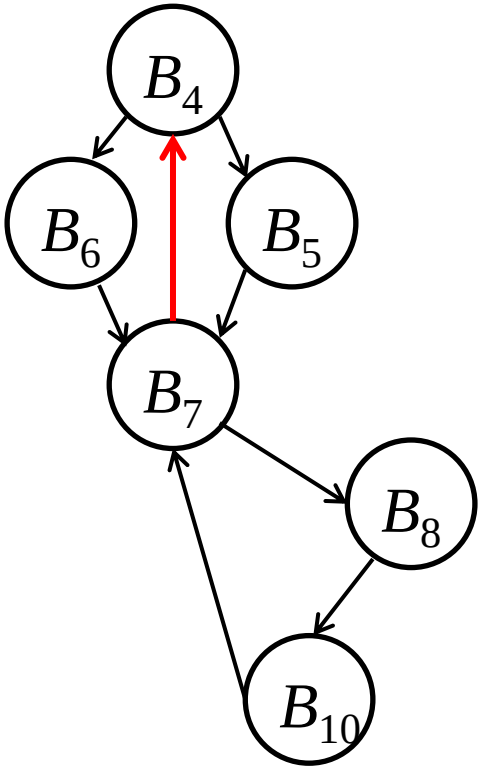
6.1.5 Пример построения естественного цикла по обратной дуге.

◊ Применим алгоритм 6.1.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$.

Отметим вершину B_4 как посещенную и выполним поиск в глубину, начиная с вершины B_7.

При этом будем считать, что на ГПУ стрелки соответствуют не концу, а началу дуги,

т.е. роль множества $Succ\ (B_7)$ выполняет множество $Pred\ (B_7) = \{B_5, B_6, B_{10}\}$

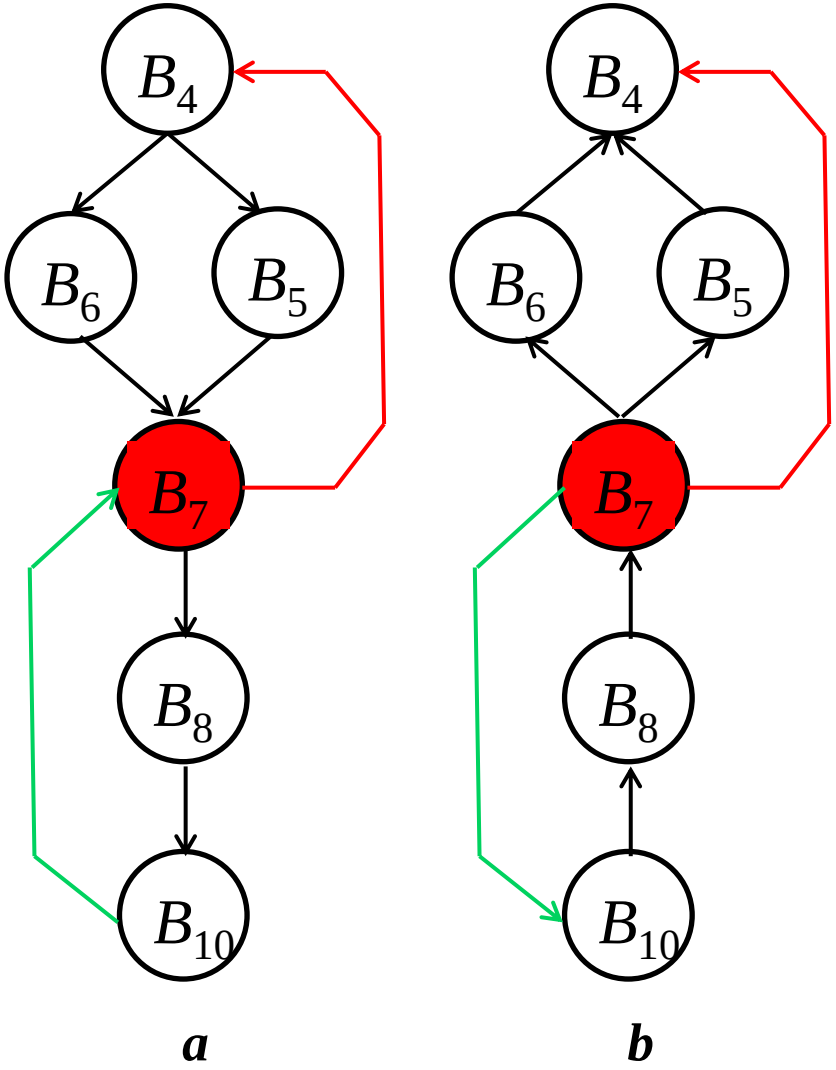


6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

◊ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$ (рисунок ***a***)

◊ Изменим направление дуг на противоположное (рисунок ***b***) и выполним поиск в глубину, начиная с вершины B_7, отметив вершину B_4 как посещенную.



6.1. Выделение естественных циклов

6.1.5 Пример построения естественного цикла по обратной дуге.

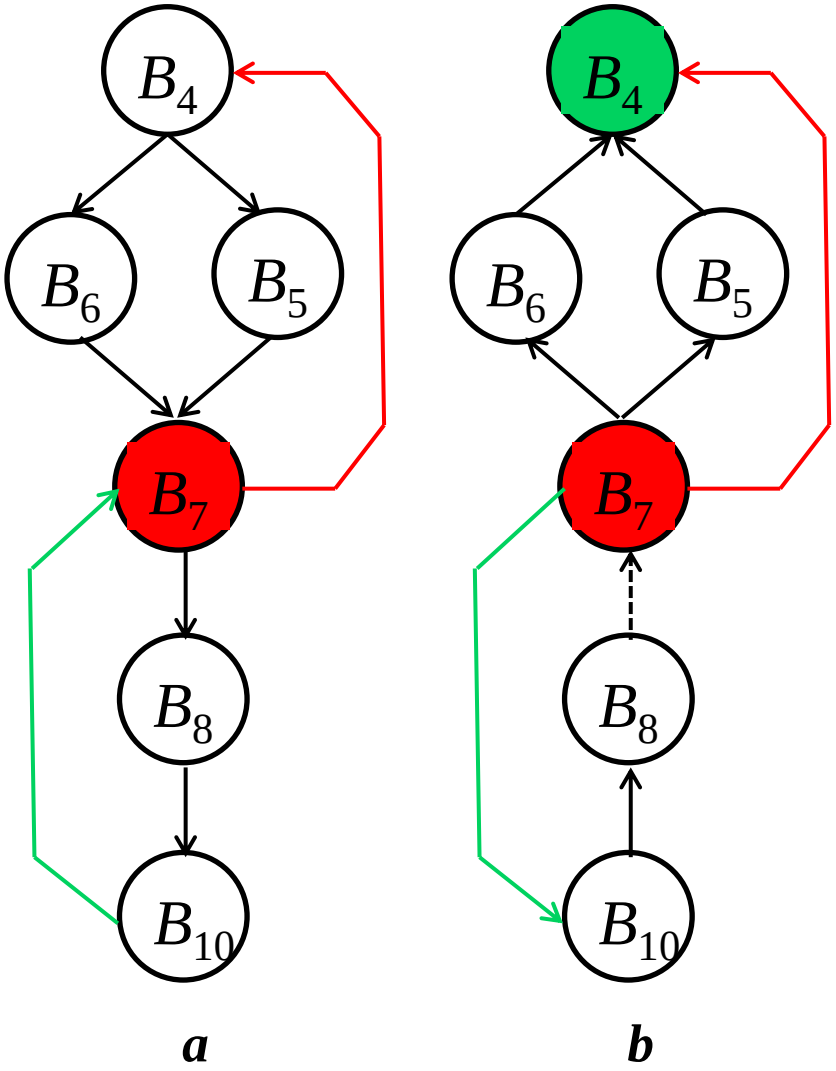
◊ Применим алгоритм 5.2.4 для построения естественного цикла, соответствующего обратной дуге $\langle B_7, B_4 \rangle$

◊ На рисунке ***b***

$Succ(B_7) = \{B_5, B_6, B_{10}\}$.

$Succ(B_{10}) = \{B_8\}$

Следовательно, в состав цикла помимо концов обратной дуги (блоков B_7 и B_4), войдут блоки B_5, B_6, B_{10}, B_8



6.2. Оптимизация циклов

6.2.1 Вводные замечания

- ◊ Оптимизация циклов существенно улучшают производительность программы в том числе и потому, что выигрыш от оптимизации реализуется на каждой итерации цикла.
- ◊ Оптимизация циклов выполняется как во время машинно-независимой, так и во время машинно-ориентированной оптимизации программы. Здесь (в части 1) рассматриваются методы машинно-независимой оптимизации циклов.
- ◊ Машинно-независимая оптимизация цикла используется для того, чтобы суметь использовать особенности алгоритма, использованного в оптимизируемой программе, для обеспечения ее высокой производительности на любом компьютере.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

- Основные методы машинно-независимой оптимизации циклов:
  - Вынесение инвариантных относительно цикла вычислений за пределы цикла.
  - Оптимизация индуктивных переменных (замена умножений более дешевыми сложениями).
  - Развертка цикла с целью уменьшить количество переходов по обратной дуге и количество проверок условия окончания цикла.
  - Инвертирование цикла: перенесение проверки на окончание цикла в конец цикла.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

- **Вынесение *инвариантных вычислений* за пределы цикла**
- На рисунке 1 мы видим пример программы, которая без необходимости повторно вычисляет инвариант цикла **a2** на каждой итерации.
- Оптимизация состоит в вынесении за пределы цикла инвариантного вычисления **a2 = a * a**, в результате чего получится программа, показанная на рисунке 2, которая эквивалентна программе с рисунка 1, но эффективнее нее, так как вычисляет значение **a2 = a * a** не **n** раз, а всего 1 раз.

```
1 void bad invariant ( ) {
2 int i, a = 4;
3 for (i = 0; i < n; ++i) {
4 int a2 = a * a;
5 /* тело цикла, в котором
6   нет инструкций, изменяющих
7   значение переменной a */
8 }
9 }
```

Рисунок 1. Повторное вычисление инварианта цикла **a2** на каждой итерации

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

- ◊ Вынесение инвариантных вычислений за пределы цикла
- ◊ На рисунке 1 мы видим пример программы, которая без необходимости повторно вычисляет инвариант цикла **a2** на каждой итерации.
- ◊ Оптимизация состоит в вынесении за пределы цикла инвариантного вычисления **a2 = a * a**, в результате чего получится программа, показанная на рисунке 2, которая эквивалентна программе с рисунка 1, но эффективнее нее, так как вычисляет значение **a2 = a * a** не **n** раз, а всего 1 раз.

```
1 void bad invariant ( ) {
2 int i, a = 4;
3 for (i = 0; i < n; ++i) {
4 int a2 = a * a;
5 /* тело цикла, в котором
6   нет инструкций, изменяющих
7   значение переменной a */
8 }
9 }
```

Рисунок 1. Повторное вычисление инварианта цикла **a2** на каждой итерации

```
1 void good invariant ( ) {
2 int i, a = 4;
3 int a2 = a * a;
4 for (i = 0; i < n; ++i) {
5 /* тело цикла, в котором
6   нет инструкций, изменяющих
7   значение переменной a */
8 }
9 }
```

Рисунок 2. Однократное вычисление инварианта цикла **a2** вне цикла.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

- ◊ Вынесение инвариантных вычислений за пределы цикла
- ◊ На рисунке 1 мы видим пример программы, которая без необходимости повторно вычисляет инвариант цикла **a2** на каждой итерации.
- ◊ Оптимизация состоит в вынесении за пределы цикла инвариантного вычисления **a2 = a * a**, в результате чего получится программа.

Целочисленное умножение в процессорах (ядрах) x86/64 выполняется в среднем за 5 тактов, так что даже для **n = 5** оптимизация сэкономит примерно 20 тактов.

```
1 void bad invariant ( ) {
2 int i, a = 4;
3 for (i = 0; i < n; ++i) {
4 int a2 = a * a;
5 /* тело цикла, в котором
6   нет инструкций, изменяющих
7   значение переменной a */
8 }
9 }
```

Рисунок 1. Повторное вычисление инварианта цикла **a2** на каждой итерации

```
1 void good invariant ( ) {
2 int i, a = 4;
3 int a2 = a * a;
4 for (i = 0; i < n; ++i) {
5 /* тело цикла, в котором
6   нет инструкций, изменяющих
7   значение переменной a */
8 }
9 }
```

Рисунок 2. Однократное вычисление инварианта цикла **a2** вне цикла.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

◊ **Индуктивные переменные**

◊ Многие циклы **`for`**, **`while_do`** и do**`_while`** имеют переменные, играющие роль счетчика циклов. Такие переменные называются *индуктивными переменными*.

Более формально: **любая переменная, значение которой изменяется на фиксированную величину при каждой итерации цикла, является *индуктивной переменной***.

◊ К индуктивным переменным можно применять два вида оптимизации:

(1) замена операции «умножение» на более быструю операцию «сложение»

(2) исключение избыточных индуктивных переменных.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

◊ Индуктивные переменные

◊ Почти все циклы **for** и некоторые циклы **while** имеют переменные, играющие роль счетчика циклов. Такие переменные называются *индуктивными переменными*.

Более формально: любая переменная, значение которой изменяется на фиксированную величину при каждой итерации цикла, является индуктивной переменной.

◊ К индуктивным переменным можно применять два вида оптимизации:

(1) замена операции «умножение» на более быструю операцию «сложение»

(2) исключение избыточных индуктивных переменных.

```
1 void bad induction var ( ) {
2 int i, j, *array ;
3 for (i = 0; i < 32; ++i) {
4 j = 4 * i;
5 /* тело цикла, использует i */
6 array[j] = rand ( );
7 }
8 }
```

Программа до оптимизации

```
1 void ros induction var ( ) {
2 int i, j, *array;
3 for(i=0, j=-4; i<32; ++i) {
4 int j += 4;
5 /* тело цикла, использует i */
6 array[j] = rand ( ) ;
7 }
8 }
```

Замена умножения на сложение

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

◊ Индуктивные переменные

◊ Почти все циклы **`for`** и некоторые циклы **`while`** имеют переменные, играющие роль счетчика циклов. Такие переменные называются *индуктивными переменными*.

Более формально: любая переменная, значение которой изменяется на фиксированную величину при каждой итерации цикла, является индуктивной переменной.

◊ К индуктивным переменным можно применять два вида оптимизации:

(1) замена операции «умножение» на более быструю операцию «сложение»

(2) исключение избыточных индуктивных переменных.

```
1 void bad induction ( ) {
2 int i, j, *array ;
3 for (i = 0; i < 32; ++i) {
4 j = 4 * i;
5 /* тело цикла, использует i */
6 arra[j] = rand ( );
7 }
8 }
```

Программа до оптимизации

```
1 void elim induction ( ) {
2 int i, *array;
3 for (i = 0; i < 32; ++i) {
4 /* тело цикла, использует i */
5 array[i * 4] = rand( );
6 }
7 }
```

Исключение избыточной индуктивной переменной **`j`**

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

◊ **Развертка цикла**

◊ Рассмотрим простую оптимизацию, называемую *разверткой цикла*. Эта оптимизация чрезвычайно проста и может применяться только к циклам с известной длиной.
Цикл с **`n`** итерациями заменяется компилятором кодом тела цикла, который просто повторяется **`n`** раз. Эта оптимизация может ускорить выполнение программы на многих процессорах, так как она **исключает любые инструкции перехода**.

◊ Отметим, что минимизация количества инструкций перехода является целью многих оптимизаций (в том числе и тех, которые не связаны с циклами), потому что эти оптимизации предотвращают возможность дорогостоящего неверного предсказания перехода. Однако, применяя эту оптимизацию, необходимо учитывать и увеличение длины программы, что, как правило, заставляет ограничиваться лишь **частичной разверткой циклов**.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

◊ **Развертка цикла**

◊ Рассмотрим простую оптимизацию, называемую *разверткой цикла*. Эта оптимизация чрезвычайно проста и может применяться только к циклам с известной длиной.
Цикл с **n** итерациями заменяется компилятором кодом тела цикла, который просто повторяется **n** раз. Эта оптимизация может ускорить

Каждая строка цикла из **n** итераций при его развертке превращается в **n** строк.
Если **n** ≤ **10**, на увеличение размеров цикла можно не обращать внимания.
А как быть, если **n** больше **10 000**?

циклами), потому что эти оптимизации предотвращают возможность дорогостоящего неверного предсказания перехода. Однако, применяя эту оптимизацию, необходимо учитывать и увеличение длины программы, что, как правило, заставляет ограничиваться лишь **частичной разверткой циклов**.

6.2. Оптимизация циклов

6.2.2 Машинно-независимая оптимизация циклов

- **Инвертирование цикла**
- Цель – уменьшить количество инструкций перехода
- *Инвертирование цикла* – это достаточно простое преобразование: цикл **`while`** преобразуется в цикл **`do-while`**, обернутый оператором **`if`**, как показано на рисунках. Это преобразование исключает 2 перехода после последней итерации цикла.

```
1 void preinversion ( ) {
2 while ( /* условие */ ) {
3 /* тело цикла */
4 }
5 }
```

Программа до оптимизации

```
1 void postinversion ( ) {
2 if ( /* условие */ ) {
3 do {
4 /* тело цикла */
5 } while ( /* условие */ );
6 }
7 }
```

Цикл после инвертирования

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.1 Код, инвариантный относительно цикла

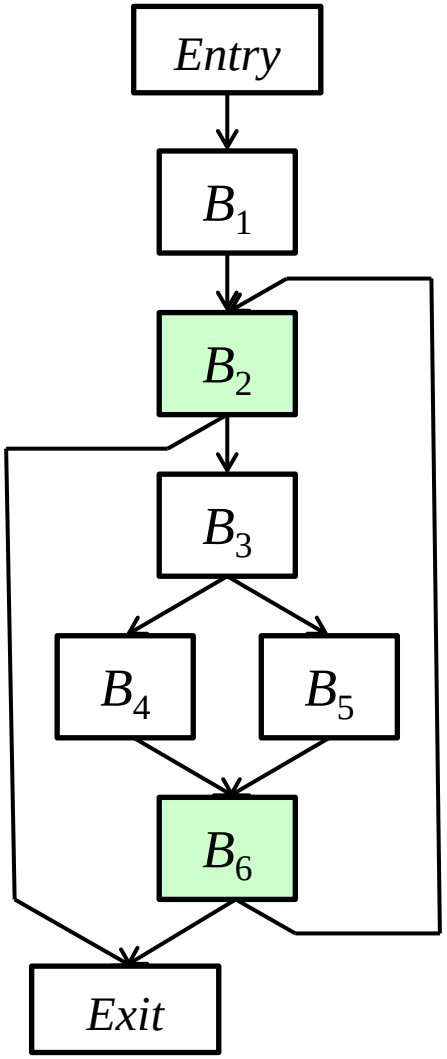
- Оптимизация состоит в том, что в ГПУ добавляется еще один базовый блок – *предзаголовок* цикла B'_2, в который выносятся из цикла все инструкции, инвариантные относительно цикла.
- Инструкция *инвариантна относительно цикла*, если все ее операнды являются инвариантами цикла.
- *Инвариантом цикла* является:
  - константа **c**;
  - переменная **u**, все определения которой находятся вне цикла;
  - переменная **v**, определяемая внутри цикла, если
    - у **v** есть всего одно определение
    - это определение находится в базовом блоке, являющемся доминатором всех выходов из цикла

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.1 Код, инвариантный относительно цикла. Пример 1

B_1 `b←2`
`i←1`
	B_2 `c←2`
	`d←b+i`
	`if(i>100)`
		B_3 `a←b+1`
		`if(i%2=0)`

B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(i<100)`

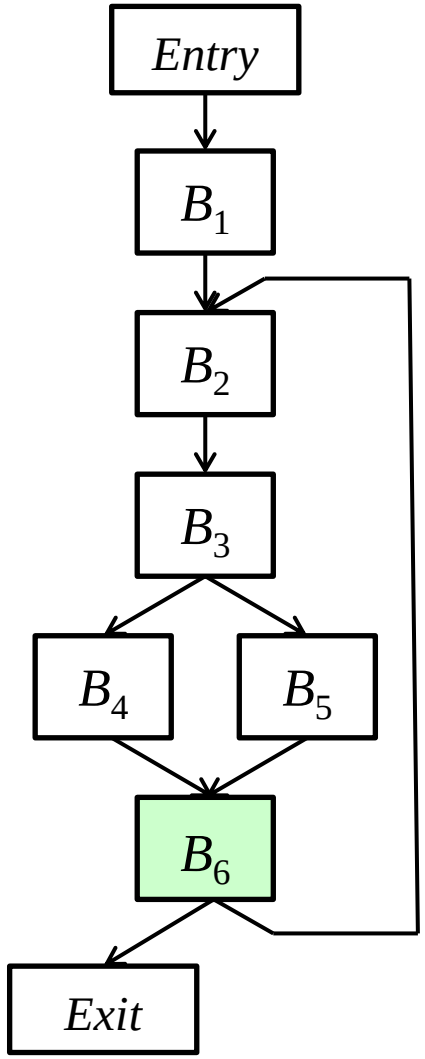


- ◊ Исходный цикл
- ◊ Блок B_1 выполняется до цикла, все остальные – в цикле.
- ◊ Выходы из цикла – блоки B_2 и B_6
- ◊ Блоки B_3, B_4, B_5 и B_6 не являются доминаторами блока B_2. Следовательно, из указанных блоков нельзя выносить код, так как вынесение кода из этих блоков не является консервативным преобразованием программы (результат программы может измениться)

6.3. Перемещение кода, инвариантного относительно цикла

6.3.1 Код, инвариантный относительно цикла. Пример 1

B_1 `b←2` `i←1`	B_2 `d←b+i` `c←2`	B_3 `a←b+1` `if(i%2=0)`
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(i<100)`



◊ Изменим исходный цикл, убрав из блока B_2 ненужное сравнение `if(i>100)`.
У цикла останется только один выход – блок B_6

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.2 Код, инвариантный относительно цикла. Пример 1

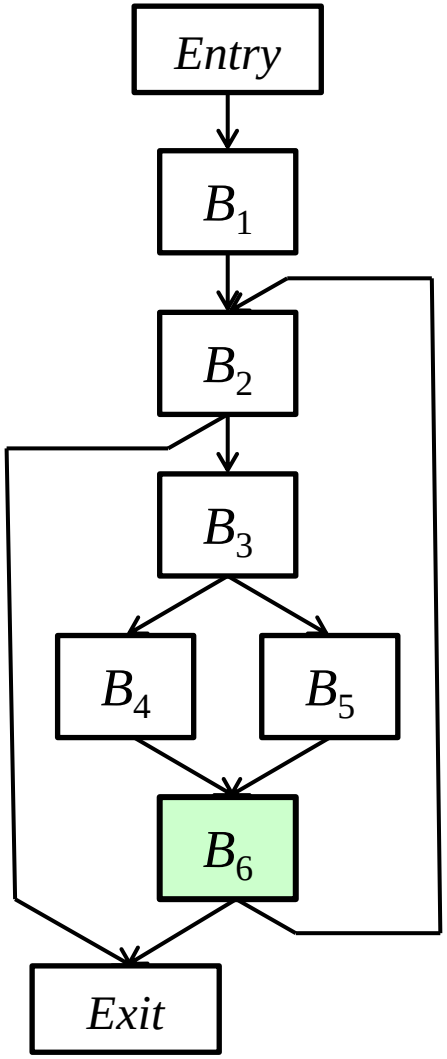
◊ Рассмотрим цикл с заголовком B_3.

B_1 `b←2` `i←1`	B_2 `d←b+i` `c←2`	B_3 `a←b+1` `if(i%2=0)`
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(i<100)`

◊ Инструкции **a←b+1, c←2**, **a<2** инвариантны относительно цикла:

a←b+1: **b** – инвариант цикла, так как его определение находится вне цикла,
1 – инвариант цикла, как константа

c←2: **2** – инвариант цикла, как константа

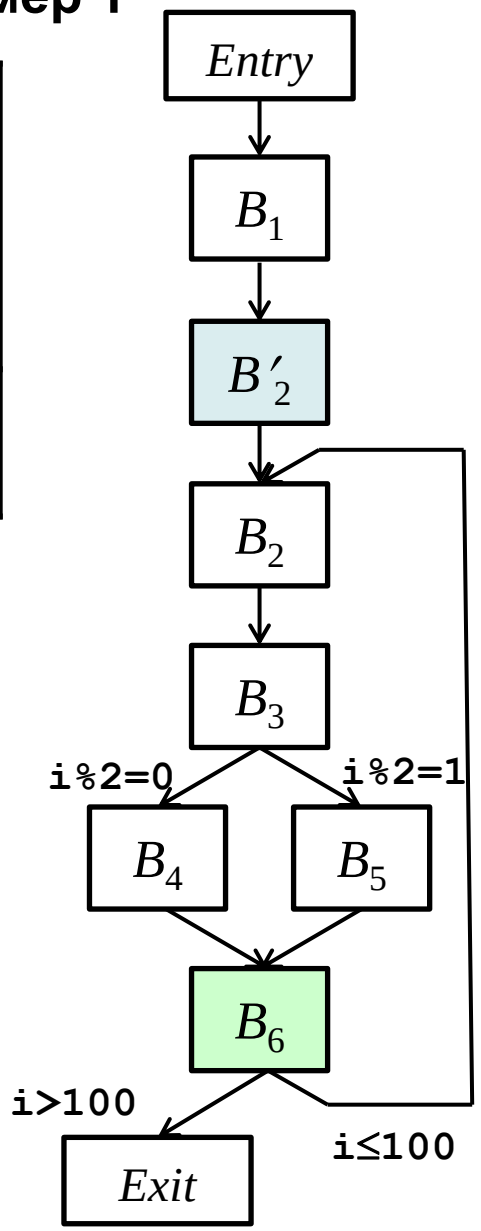


6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.2 Код, инвариантный относительно цикла. Пример 1

B_1 `b←2` `i←1`	B_2 `d←b+i` `c←2`	B'_2
	B_3 `a←b+1` `if(i%2=0)`	
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(i>100)`

◊ После добавления предзаголовка (B'_2)

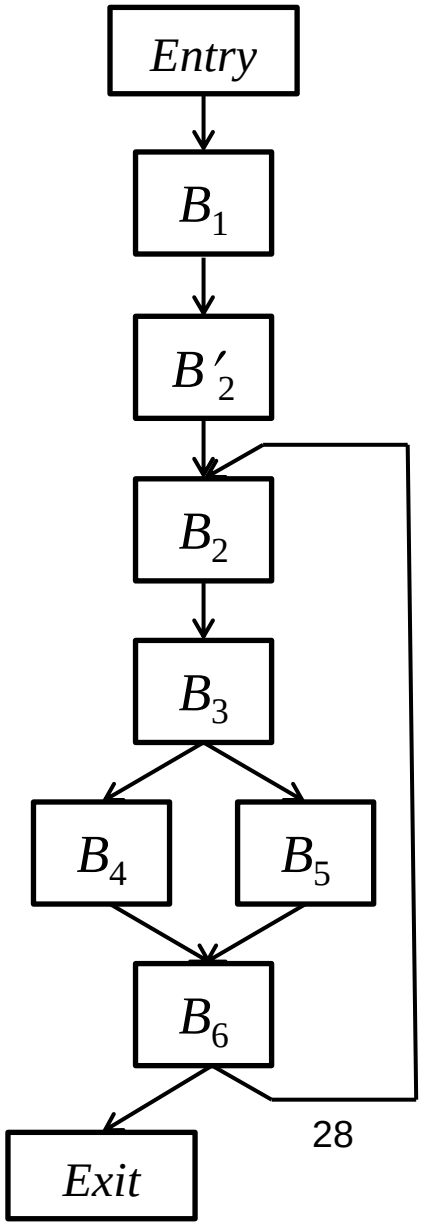


6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.2 Код, инвариантный относительно цикла. Пример 1

B_1 `b←2` `i←1`	B_2	B'_2 `a←b+1` `c←2`
	B_3 `if(i%2=0)`	
B_4 `d←a+d` `e←d+1`	B_5 `d←c` `f←d+1`	B_6 `i←i+1` `if(i>100)`

◊ После вынесения инвариантного кода в предзаголовок

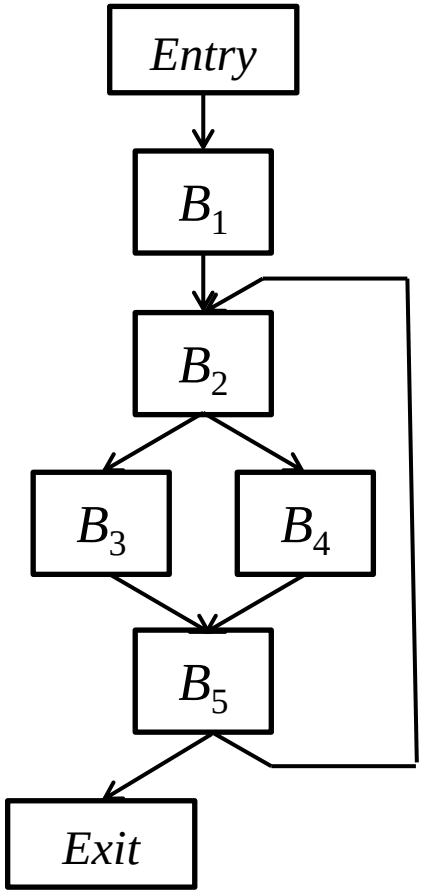


6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.3 Код, инвариантный относительно цикла. Пример 2

B_1 `b←2`
`c←3`
`i←1`
	B_2 `a←b+c`
	`if(i%2=0)`
		B_5 `d←a+1`
		`f←e+2`
		`i←i+1`
		`if(i≤100)`

B_3 `e←2`		
B_4 `e←3`		



◊ Исходный цикл

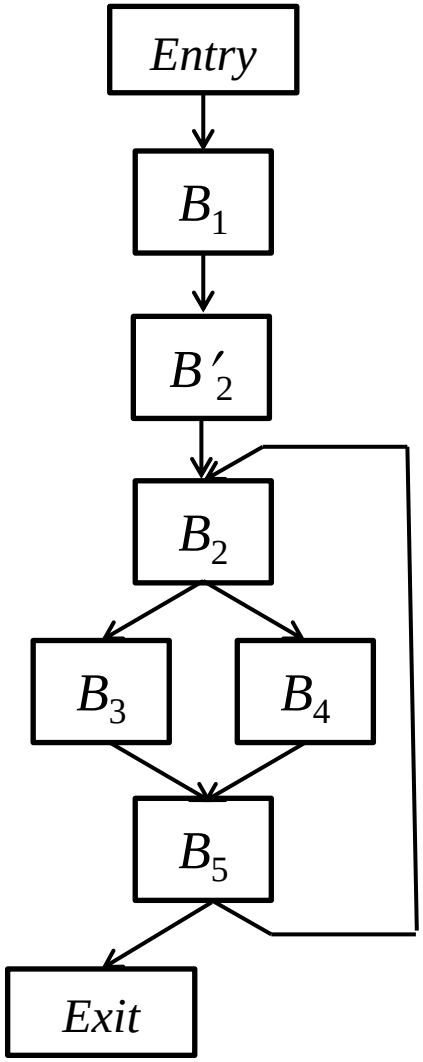
◊ Блок B_1 выполняется до цикла, все остальные – в цикле

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `a←b+c` `if(i%2=0)`	B_5 `d←a+1` `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2	
B_4 `e←3`		

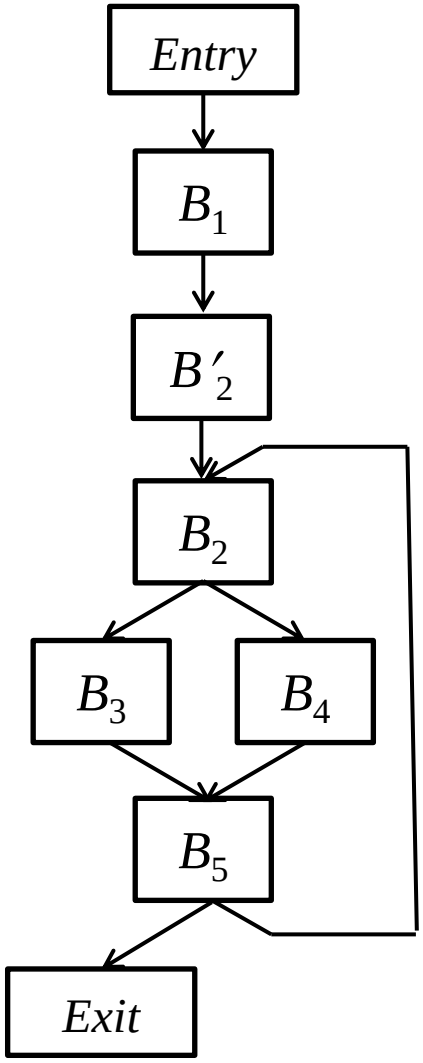
◊ После добавления предзаголовка (B'_2)



6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 **b←2** **c←3** **i←1**	B_2 **a←b+c** **if(i%2=0)**	B_5 **d←a+1** **f←e+2** **i←i+1** **if(i≤100)**
B_3 **e←2**	B'_2	
B_4 **e←3**		



◊ Инструкции **a←b+c, d←a+1** инвариантны относительно цикла:

a←b+c: определения **b** и **c** находятся вне цикла,

d←a+1: **1** константа

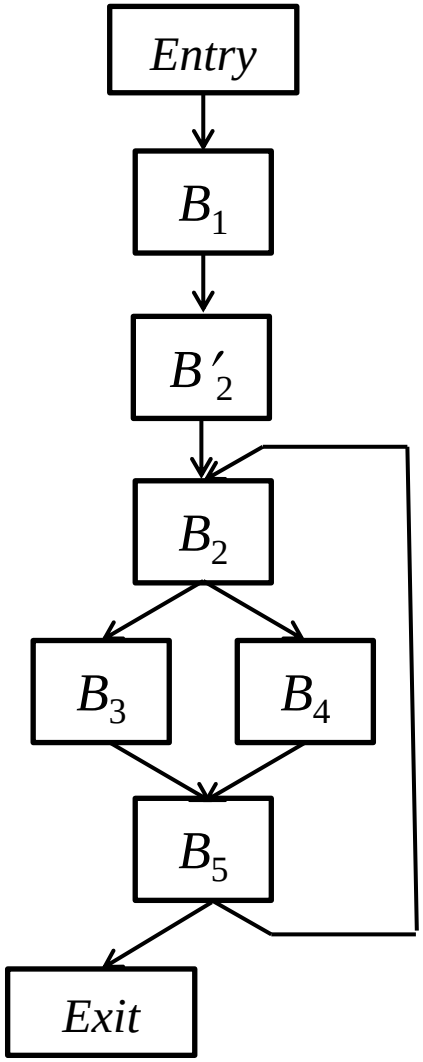
a определено только один раз в блоке , который является доминатором выхода из цикла

◊ Инструкции **e←2** и **e←3** не инвариантны относительно цикла; почему?

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `a←b+c` `if(i%2=0)`	B_5 `d←a+1` `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	
B_4 `e←3`		

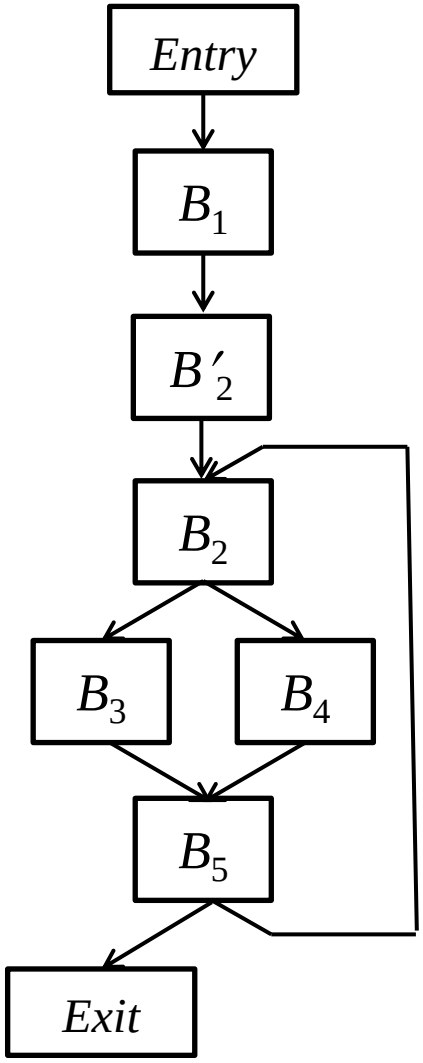


- ◊ Инструкции **e←2** и **e←3 не инвариантны относительно цикла**; почему?
- ◊ Во-первых, **e** определяется внутри цикла не один, а два раза; следовательно, значение **e** изменяется внутри цикла; при вынесении инструкций в предзаголовок **e** будет определяться только один раз и в зависимости от порядка инструкций в предзаголовке будет всегда иметь только одно значение (2 или 3)

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `a←b+c` `if(i%2=0)`	B_5 `d←a+1` `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	
B_4 `e←3`		

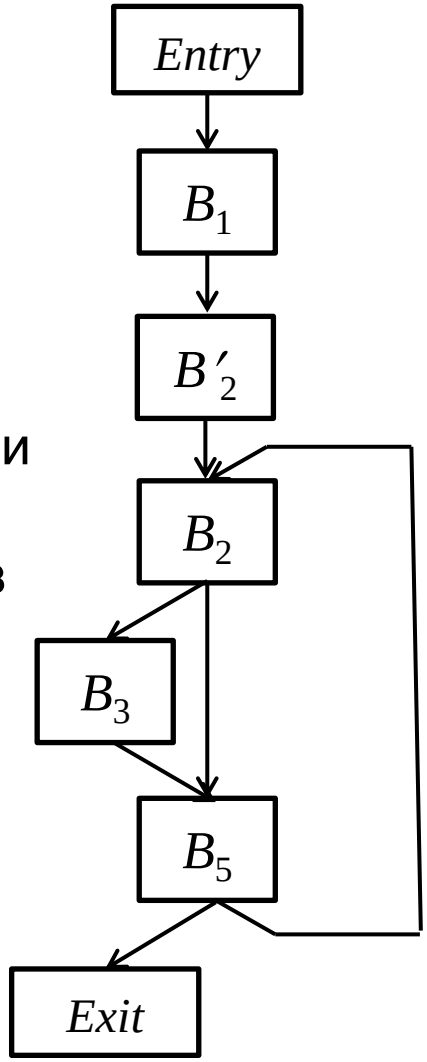


- ◊ Инструкции **`e←2`** и **`e←3`** не инвариантны относительно цикла; почему?
- ◊ Во-вторых, предзаголовок цикла является доминатором каждого базового блока, входящего в цикл, включая его заголовок (по построению); следовательно, предзаголовок является доминатором всех выходов из цикла (у рассматриваемого цикла всего один выход); значит в предзаголовок можно только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла.

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `a←b+c` `if(i%2=0)`	B_5 `d←a+1` `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	

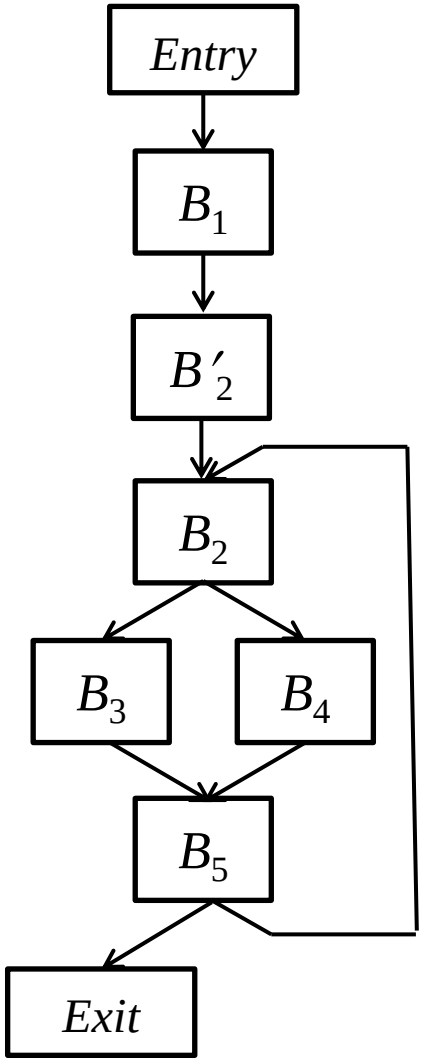


◊ В предзаголовок можно выносить только инструкции из базовых блоков являющихся доминаторами всех выходов из цикла. Если, например, удалить из цикла блок B_4, то **e** будет присваиваться только одно значение. Но даже в этом случае нельзя выносить инструкцию **e←2** в предзаголовок, так как блок B_3 все равно не будет доминатором выхода и результат выполнения программы может измениться.

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `if(i%2=0)`	B_5 `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	
B_4 `e←3`		

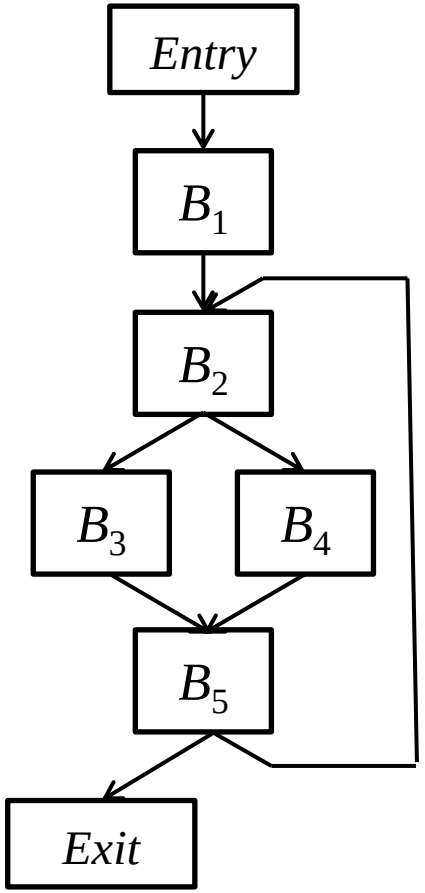


◊ Таким образом, выносить в предзаголовок можно только те инструкции, которые выполняются в блоках, доминирующих над всеми выходами из цикла (в рассматриваемом примере у цикла всего один выход) и которые выполняют единственное присваивание соответствующей переменной.

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 **b←2** **c←3** **i←1**	B_2 **if(i%2=0)**	B_5 **f←e+2** **i←i+1** **if(i≤100)**
B_3 **e←2**	B'_2 **a←b+c** **d←a+1**	
B_4 **e←3**		



Итак:

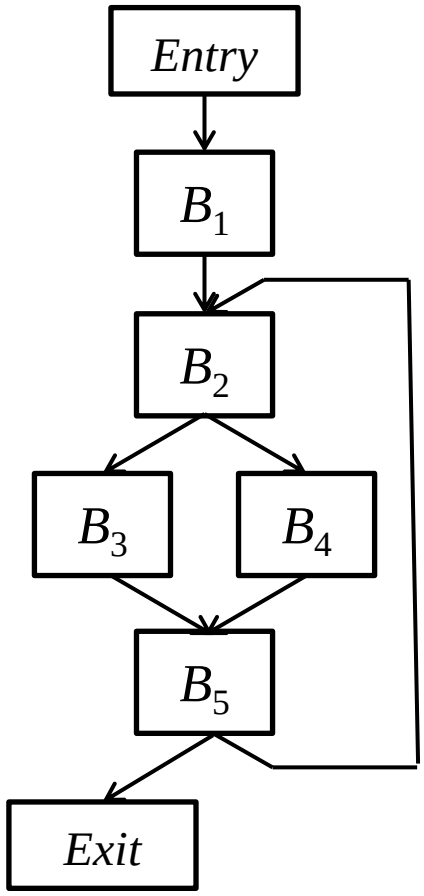
◊ **a←b+c** инвариантный код и его можно вынести в предзаголовок B'_2 (**b** и **c** определяются вне цикла, **a** определяется в блоке B_2, который доминирует над единственным выходом из цикла – блоком B_5)

◊ **d←a+1** инвариантный код и его можно вынести в предзаголовок B'_2 (**a** определяется внутри цикла только в блоке B_2, который доминирует над единственным выходом из цикла – блоком B_5)

6.3. Перемещение кода, инвариантного относительно цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `if(i%2=0)`	B_5 `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	
B_4 `e←3`		



Итак:

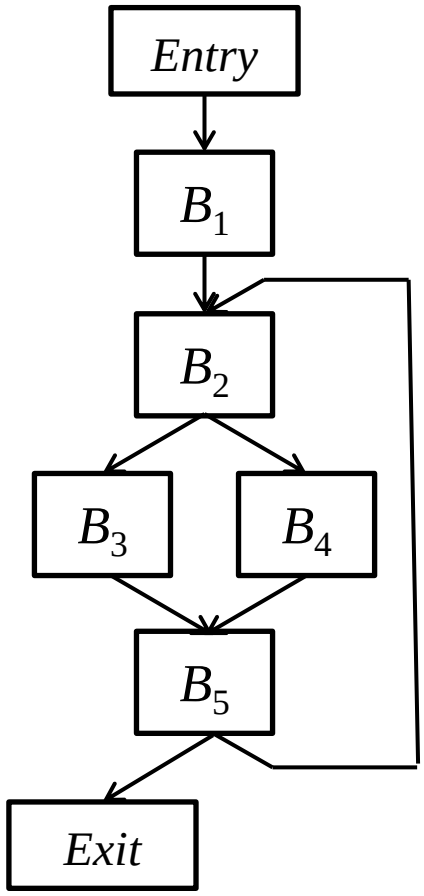
- ◊ инструкции **`e←2`** и **`e←3`** не инвариантны относительно цикла, так как выполняются в блоках, не являющихся доминаторами выхода из цикла – блока B_5
- ◊ инструкция **`f←e+2`** не инвариантна относительно цикла, так как **`e`** может изменяться во время выполнения цикла

Эти инструкции нельзя выносить в предзаголовок B'_3.

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.5 Алгоритм перемещения инвариантного кода. Пример 2

B_1 `b←2` `c←3` `i←1`	B_2 `if(i%2=0)`	B_5 `f←e+2` `i←i+1` `if(i≤100)`
B_3 `e←2`	B'_2 `a←b+c` `d←a+1`	
B_4 `e←3`		



◊ Таким образом выражение присваивания **`x = e`**, где **`e`** – инвариант цикла, можно выносить в предзаголовок цикла, если выполняются следующие три условия:

1. это единственное определение **`x`** в цикле,
3. оно является доминатором всех выходов из цикла, на которых **`x`** живо,
3. это единственное определение **`x`**, достигающее использований **`x`** внутри цикл: **`x`** не является живым на входе в заголовок цикла.

6.3 Вынесение инвариантных вычислений за пределы цикла

6.3.3 Алгоритм перемещения инвариантного кода

◊ **Алгоритм**:

1. Перед заголовком цикла вставить пустой базовый блок (будущий предзаголовок).

Для всех инструкций в теле цикла:

3. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные все операнды, у которых все определения, достигающие инструкции, находятся вне цикла
4. Отметить как инвариантные все инструкции, все операнды которых отмечены
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

6.4. Индуктивные переменные

6.4.1 Определение индуктивной переменной

◊ **Определение**. Переменная **`v`** называется *индуктивной переменной* цикла **`L`**, если на каждой итерации цикла значение **`v`** увеличивается на значение переменной (или константы) **`c`**, являющейся инвариантом цикла, то есть на каждом витке цикла выполняется инструкция:

```
v ← +, v, c
```

◊ Тривиальным примером индуктивной переменной является счетчик цикла, то есть переменная **`i`**, которой в начале цикла присваивается значение 0 (или 1) и значение которой на каждом витке цикла увеличивается на 1.

◊ Если индуктивная переменная **`v`** на каждой итерации цикла принимает значение **`c * i + d`**, где **`c`** и **`d`** – инварианты цикла, то **`v`** является линейной индуктивной переменной.

◊ Когда в цикле удается обнаружить индуктивные переменные, становится возможным выполнить различные оптимизации этого цикла.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

◊ Нетрудно доказать, что линейные функции от индуктивных переменных тоже индуктивные переменные.
Следовательно, в циклах, могут встречаться семейства индуктивных переменных. Обычно в циклах бывает несколько таких семейств.

◊ *Основной* индуктивной переменной по определению является индуктивная переменная **`i`**, все определения которой эквивалентны определению вида **`i ← +, i, c`**, где **`c`** – инвариант цикла (как правило, константа). Нет необходимости, чтобы значение c было одинаковым в каждом таком определении.
Основная индуктивная переменная является *линейной*, если в цикле имеется всего одно ее определение, причем это определение является доминатором (или постдоминатором) всех остальных вершин цикла.

◊ *Производная* индуктивная переменная **`j`** выражается через одну из основных индуктивных переменных **`i`** как **`c * i + d`**, где c и d – инварианты цикла.

◊ Основная индуктивная переменная **`i`** и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.

6.4. Индуктивные переменные

6.4.2 Семейства индуктивных переменных

◊ Основная индуктивная переменная **i** и все ее производные индуктивные переменные составляют *семейство индуктивных переменных*.
Для производной индуктивной переменной **j = c * i + d** удобно использовать обозначение **j = ⟨i, c, d⟩**.
Применяя это обозначение к **i**, получим **i = ⟨i, 1, 0⟩**

◊ В примере справа:

- **i** – основная индуктивная переменная;
- **j** – линейная основная индуктивная переменная;
- **k** и **l** – линейные производные индуктивные переменные семейства **j**;
- **m** – производная индуктивная переменная семейства **i**.

```
while (i < 10) {
j = j + 2;
if (j > 4)
      i = i + 1;
i = i - 1;
k = j + 10;
l = k * 4;
m = i * 8;
}
```

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- Чтобы начать анализ потока данных необходимо построить начальное отображение (**`var`**, **`val`**), где **`var`** – переменная, а **`val`** – ее значение. Эти пары помещаются в очередь **`Worklist`**.
  - Сначала находят **основные** индуктивные переменные. Для этого просматриваются все инструкции цикла и находятся все переменные **`i`**, которым сначала присваивается целое значение **`c`** (**`i ← c`**), а потом ее значения изменяются только инструкциями вида **`i ← +, i, d`**, или **`i ← +, d, i`**, где **`d`** – инвариант цикла.
  - Отметим, что присваивание **`i ← c`** должно быть **только одно.** Если таких присваиваний несколько, т. е. есть **`n`** инструкций **`i ← c`**$_k$ с разными **`c`**$_k$, (**`k = 1, … , n`**), то переменная **`i`** **расщепляется** на **`k`** переменных **`i`**$_k$.
  - Значением основной индуктивной переменной **`i`**, т. е. **`val(i)`** является «тройка» ⟨**`i,1,0`**⟩, так как очевидно, что **`i = 1*i + 0`**. Каждой основной индуктивной переменной **`i`** соответствует пара (**`i`**, ⟨**`i,1,0`**⟩), которая помещается в **`Worklist`**.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

◊ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.

- Если обнаруживаются инструкции вида
  p ← +, j, c или **q ← *, j, c**,
  где **j** – индуктивная переменная семейства **i**: **val(j) = ⟨i,a,b⟩**, а **c** – инвариант цикла, то **p** и **q** – тоже индуктивные переменные семейства **i**, причем
  val(p) = ⟨i,a,b+c⟩ и **val(q) = ⟨i,a*c,b*c⟩**.
- Если обнаруживаются инструкции вида **p ← e**, где **e** – выражение, не содержащее индуктивных переменных, то **val(p) = NIV**
- Всем остальным переменным сопоставляется значение ⊤ (**Undef**)
- Все пары (**var**, **val**), найденные на первом этапе, кроме пар, у которых **val** = **NIV** помещаются в том порядке, в котором они были найдены в очередь **Worklist**.
- На этом первый этап алгоритма обнаружения индуктивных переменных заканчивается.

6.4. Индуктивные переменные

6.4.3 Обнаружение индуктивных переменных

- ◊ После того, как начальное отображение построено и **`Worklist`** заполнен, начинается второй этап алгоритма обнаружения индуктивных переменных.
  - ⟠ Из очереди **`Worklist`** выбирается очередная пара (**`var, val`**)
  - ⟠ Если эта пара имеет вид (**`i`**, ⟨**`i,1,0`**⟩), она помещается в список найденных индуктивных, начиная семейство **`i`**.
    Теперь все индуктивные переменные, значение которых имеет первым членом «тройки» **`i`**, будут помещаться в это семейство.
  - ⟠ Если эта пара имеет вид (**`j`**, ⟨**`i,a,b`**⟩), выполняется обход ГПУ, чтобы выяснить, не превратится ли **`val(j)`** в **`NIV`**. Каждая пара анализируется независимо от других пар. Это связано со структурой полурешетки. В основном уточняются кандидаты в индуктивные переменные, имеющие значение **`Undef`**.
  - ⟠ Анализ состоит в том, чтобы просмотреть все базовые блоки, в которых встречается **`j`** и убедиться, что во всех блоках **`val(j)`** остается в своем семействе индуктивных переменных. Основную роль при таком анализе играет операция сбора для функции **`val`**.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций

◊ Будет показано, что для индуктивной переменной можно заменить умножение сложением (т. е. арифметическую операцию, выполняемую более сложным (и долгим) алгоритмом заменить на более простую и быструю). Причем при указанной замене количество выполняемых операций (инструкций) не изменится.

◊ Рассмотрим простейший пример:

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

Значения переменной **j** вычисляются с использованием умножения, но **j** – производная индуктивная переменная ⟨**i,3,a**⟩, принадлежащая семейству основной индуктивной переменной **i**.

◊ Как вычислять **j**, используя сложение вместо умножения?

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

◊ Как вычислять **j**, используя сложение вместо умножения?

◊ Сделать это очень просто:
Для производной индуктивной переменной **j** из семейства основной индуктивной переменной **i**: **val(j)** = ⟨**i,c,d**⟩ необходимо выполнить следующий простой **алгоритм**:

1. Создать новые переменные **s** и **k**, и в предзаголовке цикла выполнить присваивания **s = c*i + d;** и **k = c*h;**
2. В теле цикла заменить определение **j = e** на **j = s;**
3. В теле цикла после присваивания **i = i + h** вставить присваивание **j = j + k**;

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, продолжение примера

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

◊ Применив алгоритм к рассматриваемому циклу, получим

```
i = 1;
s = p + 3*i;
while (i < a.length) {
      j = s;
      a[j] = a[j] + 1;
      i = i + 2;
      s = s + 6;
}
```

◊ И, что интересно, в преобразованном цикле нет явной зависимости между **j** и **i**: счетчик цикла и индекс элемента массива изменяются как бы независимо.

6.4. Индуктивные переменные

6.4.4 Снижение сложности операций, окончание примера

◊ Таким образом, алгоритм снижения стоимости позволил заменить в теле цикла

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

умножение (`j = p + 3*i`) на сложение (`s = s + 6`),

в результате чего получился цикл

```
i = 1;
s = p + 3*i;        //предзаголовок цикла
while (i < a.length) {
      j = s;
      a[j] = a[j] + 1;
      i = i + 2;
      s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

◊ Рассмотрим оптимизированный цикл

```
i = 1;
s = p + 3*i;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ В нем три индуктивных переменных **`i`**, **`j`** и **`s`**.
Анализ покажет, что две из них лишние, и их можно исключить.

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ **Во-первых**, переменные **j** и **s** – это, по существу, одна переменная, так как **s** используется только для приращения индекса **j**.
Эти переменные (**j** и **s**) можно объединить в одну, например **s** и исключить лишнюю индуктивную переменную **j**.
Цикл после этого будет иметь вид (лишняя индуктивная переменная **j** исключена):

```
i = 1;
s = p + 3*i;
while (i < a.length) {
        a[s] = a[s] + 1;
        i = i + 2;
        s = s + 6;
}
```

6.4. Индуктивные переменные

6.4.5 Исключение (лишних) индуктивных переменных

```
i = 1;
s = p + 3*i;
while (i < a.length) {
        a[s] = a[s] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ **Во-вторых**, переменная `i` используется только в качестве счетчика цикла. Такие переменные называются *почти бесполезными*.
Для того, чтобы исключить `i`, нужно вычислить в предзаголовке верхнюю границу `t` для `s`.
Вспомнив, что `s` – производная индуктивная переменная ⟨`i,3,a`⟩ семейства `i`, получим **`t = a + 3*a.length`**

◊ После исключения `i` цикл примет окончательный вид:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
        a[s] = a[s] + 1;
        s = s + 6;
}
```

◊ Предзаголовок можно не оптимизировать: он выполняется всего один раз.

6.4. Индуктивные переменные

6.4.5 Исключение индуктивных переменных

◊ Таким образом, снижение стоимости и исключение (лишних) индуктивных переменных позволили оптимизировать исходный цикл

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

◊ Получив следующий оптимизированный цикл:

```
i = 1;
s = p + 3*i;
t = 3*a.length
while (s < t) {
      a[s] = a[s] + 1;
      s = s + 6;
}
```

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

◊ В языках *со строгой типизацией* (например, в языке *Java*), доступ к элементу массива по индексу возможен только тогда, когда индекс элемента не выводит за границы массива. Следовательно, транслятор *Java*-программы на байт-код, а значит и *JIT* должны вставлять в циклы соответствующие проверки: прежде, чем вычислить адрес элемента **`a[j]`**, компилятор должен убедиться, что **`j`** – безопасный индекс, не выводящий за пределы массива **`a[]`**.

◊ В частности, в цикл из примера 6.4.4 будет добавлена проверка, которая гарантирует, что индекс **`j`**, по которому производится доступ, удовлетворяет условиям.

$$0 \le j < a.length.$$

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

```
i = 1;
while (i < a.length) {
      j = p + 3*i;
      a[j] = a[j] + 1;
      i = i + 2;
}
```

◊ В цикл, рассматриваемый в примере 6.4.4, добавится строка (она выделена цветом):

```
i = 1;
while (i < a.length) {
  j = p + 3*i;
  if (i < 0 | i ≥ a.length) goto L_err;
        a[j] = a[j] + 1;
        i = i + 2;
  }
```

◊ Наличие указанной строки внутри цикла существенно замедлит его выполнение (ведь на каждом витке цикла добавится два сравнения).

6.5. Исключение проверок границ массивов

6.5.1 В чем проблема

◊ Как показывает опыт программирования на языках *со слабой типизацией* (например, на любимом языке физиков *Fortran*), где такие проверки должен вставлять программист (если сочтет нужным), исключение таких проверок повышало скорость выполнения программы больше чем на порядок, но…
одновременно помогало хакерам и их вирусам проникать внутрь компьютера на котором программа выполнялась.

◊ Итак, **проблема в том, чтобы научиться удалять из цикла хотя бы часть проверок границ массивов, не уменьшая безопасности программы**.

6.5. Исключение проверок границ массивов

6.5.2 Решение проблемы

◊ Можно реализовать оператор

```
if (0 < i & i < n)
```

с помощью одной проверки (а не двух).

◊ Предположим, что **n** – положительное целое со знаком (**int n > 0**), а **i** – целое со знаком (**int i**). Тогда единственная проверка

```
(unsigned)i < n
```

обеспечит и **i < n**, и **i > 0**, так как при переводе отрицательного **i** в тип **unsigned**, **i** получит очень большое положительное значение. Отметим, что многие современные процессоры имеют операцию «беззнаковое сравнение», так что все это можно реализовать. Примером является операция **JAE** (**Jump Above or Equal**) в процессорах *Intel*.

6.5. Исключение проверок границ массивов

6.5.2 Решение проблемы.

- Лучше всего, конечно, вообще исключить проверку, или, по крайней мере, вынести ее в предзаголовок.
- Идея в том, что выполнение условия выхода из цикла (в рассматриваемом примере **`i < a.length`**) часто гарантирует, что обращения к массиву не выводят за его пределы.
- Если в этом можно убедиться во время статического анализа, проверка может быть удалена.
- Если это выясняется во время динамического анализа (или во время выполнения программы), цикл может быть представлен в двух версиях:
  - Быстрой, в которой нет проверок границ и
  - Медленной, в которой такие проверки есть.

  А пользователь пусть сам разбирается, где и когда какую версию использовать.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◊ Рассмотрим проблему исключения проверок границ массивов, когда выполняются следующие условия:

1. Индуктивная переменная **`j`** семейства **`i`** сравнивается с инвариантом цикла **`u`** (проверка условия выхода из цикла **`j < u)`**.
2. Индуктивная переменная **`k`** того же семейства **`i`** сравнивается с инвариантом цикла **`n`** (проверка условия выхода из цикла **`k < n`**), причем из **`j < u`** следует **`k < n`**.
3. **`k`** и **`j`** изменяются в одном и том же направлении.
4. Проверка условия **`j < u`** находится в блоке, являющемся доминатором блока, в котором находится проверка условия **`k < n`**.

Утверждение. При выполнении условий 1, 2, 3 и 4 проверка условия **`k < n`** избыточна и может быть удалена.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◊ Для обоснования утверждения, выясним, когда из $j < u$ следует $k < n$?

Пусть $j = \langle i, c_j, d_j \rangle$, $k = \langle i, c_k, d_k \rangle$.

Если условие $j < u$ выполняется, то

$$c_j * i + d_j < u$$

Без потери общности можно считать, что $c_j > 0$. Тогда

$$i < (u - d_j) / c_j$$

Следовательно,

$$k = c_k * i + d_k < c_k * \left((u - d_j) / c_j \right) + d_k$$

Если удастся показать в результате статического или динамического анализа, что правая часть последнего неравенства $\leq n$, то условие $k < n$ будет выполнено.

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

◊ Итак, необходимо выяснить, верно ли что

$$c_k * \left((u - d_j) / c_j \right) + d_k \le n \quad (*)$$

Это можно либо установить во время компиляции, либо (если статический анализ недостаточен) попытаться поместить соответствующую проверку в предзаголовок цикла.

```
s = p + 3*i;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ В нашем примере условие **`i < a.length`** оказывается достаточным для одновременной проверки и верхней границы массива **`a[]`**. В этом нетрудно убедиться во время компиляции подставив $c_i = 1$, $d_i = 0$, вместо (c_j, d_j) и (c_k, d_k) в формулу (*). Получим верное неравенство:

`1*(a.length - 0)/1 + 0 ≤ a.length`, или
`a.length ≤ a.length`.

Что касается 0, левой границы для **`i`**, то проверку условия **`i > 0`** можно вынести в предзаголовок (так как 0 – инвариант цикла).

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

```
s = p + 3*i;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ В итоге, получим следующий цикл, в котором, конечно же, можно выполнить оптимизации, связанные с удалением лишних индуктивных переменных.

```
s = p + 3*i;
if (i < 0) goto L_err;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

L_{err} – метка, начиная с которой помещен код обработки исключения «выход за границы массива».

6.5. Исключение проверок границ массивов

6.5.3 Использование индуктивных переменных.

```
s = p + 3*i;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

◊ В итоге, получим следующий цикл, в котором, конечно же, можно выполнить оптимизации, связанные с удалением лишних индуктивных переменных.

```
s = p + 3*i;
if (i < 0) goto L_err;
while (i < a.length) {
        j = s;
        a[j] = a[j] + 1;
        i = i + 2;
        s = s + 6;
}
```

Вместо двух проверок внутри цикла, одна простая в предзаголовке

L_{err} – метка, начиная с которой помещен код обработки исключения «выход за границы массива».

6.6. Раскрутка циклов

6.6.1 Что это такое

- **Простой пример**. Выполним раскрутку внутреннего цикла в гнезде.
  - Исходное гнездо циклов (на языке С):

```
for(j = 1; j <= nj; j++){
    for(i = 1; i <= ni; i++){
        y[i] += x[j] * m[i][j];
    }
}
```

  - После раскрутки внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++){
    mod_i = ni % 4;
    if(mod_i >= 1){
        for(i = 1; i <= mod_i; i++){
            y[i] += x[j] * m[i][j];
        }
    for(i = mod_i + 1; i <= ni; i += 4){
        y[i] += x[j] * m[i][j];
        y[i+1] += x[j] * m[i+1][j];
        y[i+2] += x[j] * m[i+2][j];
        y[i+3] += x[j] * m[i+3][j];
    }
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое

◊ **Простой пример**. Выполним раскрутку внутреннего цикла в гнезде.

- Исходное гнездо циклов (на языке С):

```
for(i = 1; i <= ni; i++){
```

Раскрутка цикла

копирует тело цикла для нескольких итераций и корректирует вычисление индексов соответствующим образом.

- После раскрутки внутреннего цикла на 4:

```
for(j = 1; j <= nj; j++){
    mod_i = ni % 4;
    if(mod_i >= 1){
```

Если **ni** не делится на 4, то остается кусок цикла, который выполняется в коротком *цикле-прологе*.

Назначение цикла-пролога – гарантировать, что количество итераций цикла раскручиваемого на m, кратно m.

Такое преобразование цикла (расщепление цикла для обеспечения удобных границ) называется *выравниванием цикла*.

```
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое.

- **Простой пример**. Но можно выполнить и **раскрутку внешнего цикла**.
  - После раскрутки внешнего цикла на 4:

```
mod_j = nj % 4;
if(mod_j >= 1){
        for(j = 1; j <= mod_j; j++){
                for(i = 1; i < ni; i++){
                        y[i] += x[j] * m[i][j];
                }
        }
}
for(j = mod_j + 1; j <= nj; j += 4){
        for(i = 1; i < ni; i++){
                y[i] += x[j] * m[i][j];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+1] * m[i][j+1];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+2] * m[i][j+2];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+3] * m[i][j+3];
        };
}
```

6.6. Раскрутка циклов

6.6.1 Что это такое.

◊ **Простой пример**. Но можно

◈ После раскрутки вне

```
for(j = 1; j < nj; j++){
        for(i = 1; i < ni; i++){
                y[i] += x[j] * m[i][j];
        }
}
```

```
mod_j = nj % 4;
if(mod_j >= 1){
        for(j = 1; j <= mod_j; j++){
                for(i = 1; i < ni; i++){
                        y[i] += x[j] * m[i][j];
                }
        }
}
for(j = mod_j + 1; j <= nj; j += 4){
        for(i = 1; i < ni; i++){
                y[i] += x[j] * m[i][j];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+1] * m[i][j+1];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+2] * m[i][j+2];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+3] * m[i][j+3];
        };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

- Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
  Слияние корректно, когда каждое определение и каждое использование в результирующем (слитом) цикле имеют такие же значения, что и в исходных (сливаемых) циклах.
- В рассматриваемом простом примере можно выполнить три слияния циклов.
  - До слияния внутренних циклов:

```
for(j = mod_j + 1; j <= nj; j += 4){
        for(i = 1; i < ni; i++){
                y[i] += x[j] * m[i][j];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+1] * m[i][j+1];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+2] * m[i][j+2];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+3] * m[i][j+3];
        };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

- Слияние циклов – объединение двух циклов с одинаковыми границами изменения индекса и шагом в один.
- В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.
  - После первого слияния:

```
for(j = mod_j + 1; j <= nj; j += 4){
        for(i = 1; i < ni; i++){
                y[i] += x[j] * m[i][j];
                y[i] += x[j+1] * m[i][j+1];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+2] * m[i][j+2];
        };
        for(i = 1; i < ni; i++){
                y[i] += x[j+3] * m[i][j+3];
        };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

- В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.
  - После третьего слияния:

```
for(j = mod_j + 1; j <= nj; j += 4){
  for(i = 1; i < ni; i++){
    y[i] += x[j] * m[i][j];
    y[i] += x[j+1] * m[i][j+1];
    y[i] += x[j+2] * m[i][j+2];
    y[i] += x[j+3] * m[i][j+3];
  };
}
```

  - Последний цикл можно преобразовать к виду:

```
for(j = mod_j + 1; j <= nj; j += 4){
  for(i = 1; i < ni; i++){
    y[i] = y[i] + x[j] * m[i][j] + x[j+1] * m[i][j+1]
           + x[j+2] * m[i][j+2] + x[j+3] * m[i][j+3];
  };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

◊ В рассматриваемом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◊ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;
if(mod_j >= 1){
   for(j = 1; j <= mod_j; j++){
      for(i = 1; i < ni; i++){
         y[i] += x[j] * m[i][j];
      }
   }
},
```

получаем окончательно следующий цикл:

```
for(j = mod_j + 1; j <= nj; j += 4){
   for(i = 1; i < ni; i++){
      y[i] += x[j] * m[i][j];
      y[i] += x[j+1] * m[i][j+1];
      y[i] += x[j+2] * m[i][j+2];
      y[i] += x[j+3] * m[i][j+3];
   };
}
```

6.6. Раскрутка циклов

6.6.2 Слияние циклов.

◊ В рассматриваемом простом примере можно в последнем цикле выполнить три слияния внутренних циклов.

◊ Таким образом, оптимизируя цикл

```
mod_j = nj % 4;
if(mod_j >= 1){
```

Рассмотренное оптимизирующее преобразование называется *раскруткой с последующим сжатием.*

```
    }
  }
```

Если у сливаемых циклов границы изменения переменной цикла i не совпадают, можно применить *выравнивание* циклов.

```
for(j = mod_j + 1; j <= nj; j += 4){
  for(i = 1; i < ni; i++){
    y[i] += x[j] * m[i][j];
    y[i] += x[j+1] * m[i][j+1];
    y[i] += x[j+2] * m[i][j+2];
    y[i] += x[j+3] * m[i][j+3];
  };
}
```

6.6. Раскрутка циклов

6.6.3 Сравнение двух видов раскрутки циклов.

◊ Раскрутка внутреннего цикла производит код, **выполняющий намного меньше проверок на выход из цикла**. Доступ к двумерному массиву `m[i][j]` последователен, так как С-массив располагается по строкам.

◊ Раскрутка внешнего цикла не только сокращает количество проверок на выход из цикла, но и (особенно в случае раскрутки с последующим сжатием) обеспечивает повторное использование `y[i]`, а также последовательный доступ как к элементам `x`, так и к элементам `m`. Увеличение повторного использования данных существенно изменяет соотношение между арифметическими операциями и операциями доступа к памяти в цикле, повышая *локальность* данных.

◊ Кроме того, при каждом подходе могут проявляться и другие прямые и косвенные улучшения кода. Окончательная производительность цикла зависит от всех улучшений, как прямых, так и косвенных.

◊ Важным косвенным улучшением помимо увеличения локальности данных является увеличение количества операций в теле цикла.

◊ Раскрутка цикла позволяет реализовать его параллельное выполнение (этот вопрос будет рассмотрен при планировании кода).

6.7. Заключительные замечания

6.7.1 Изменение порядка циклов в гнезде.

◊ Рассмотрим цикл:

```
for (int j = 0; j < 3; j++){
    for (int i = 0; i < 2; i++){
        a[i][j] = i + j;
   }
}
```

◊ Двумерный массив $\begin{Vmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{Vmatrix}$

хранится в памяти как

$$\left| a_{00}\ a_{01}\ a_{02}\ a_{03} \right| \left| a_{10}\ a_{11}\ a_{12}\ a_{13} \right| \left| a_{20}\ a_{21}\ a_{22}\ a_{23} \right|$$

◊ Цикл выполняется сначала для a_{00}, потом для a_{10}, потом для a_{20} потом для a_{01}, потом для a_{11} и так далее. То есть массив **a** будет обходиться с шагом 4.

6.7. Заключительные замечания

6.7.1 Изменение порядка циклов в гнезде.

◊ Если изменить порядок циклов:

```
for (int i = 0; i < 2; i++){
    for (int j = 0; j < 3; j++){
        a[i][j] = i + j;
   }
}
```

◊ двумерный массив $\begin{Vmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{Vmatrix}$

который хранится в памяти как

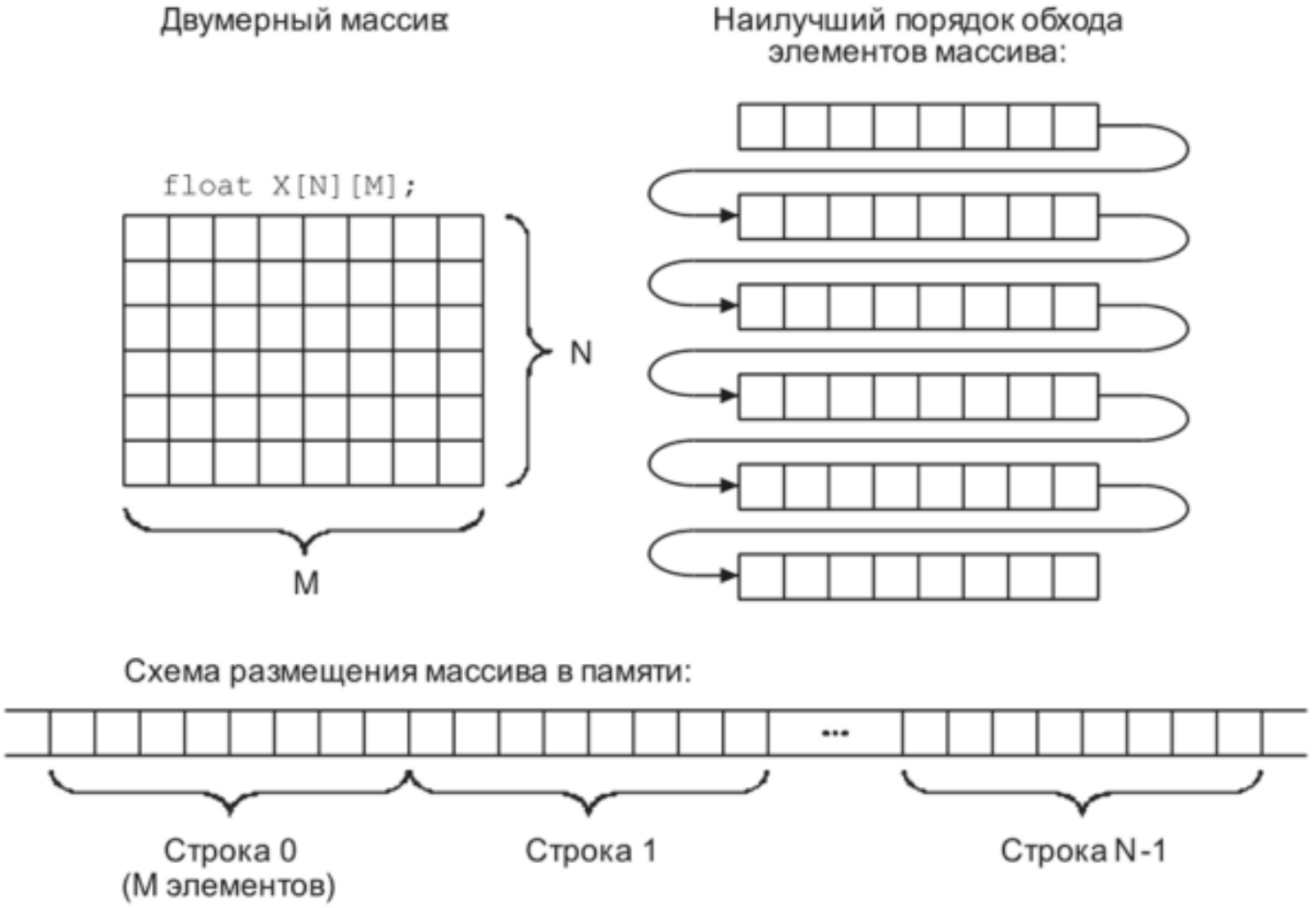
$$\left\| a_{00} \;\; a_{01} \;\; a_{02} \;\; a_{03} \right\| \left\| a_{10} \;\; a_{11} \;\; a_{12} \;\; a_{13} \right\| \left\| a_{20} \;\; a_{21} \;\; a_{22} \;\; a_{23} \right\|$$

будет обходиться с шагом 1.

6.7. Заключительные замечания

6.7.1 Изменение порядка циклов в гнезде.

Если бы у меня хватило терпения на комментирование этого варианта (см. рисунок), результаты получились бы более впечатляющими.



6.7. Заключительные замечания

6.7.2 Другие преобразования циклов.

- ◊ В компиляторах применяются и другие преобразования циклов. Но эти преобразования применяются во время планирования кода для обеспечения параллельного выполнения инструкций программы на каждом ядре процессора. Эти преобразования будут рассмотрены во время изучения темы 17 «Оптимизация циклов (часть II)».

- ◊ В частности будут рассмотрены преобразования циклов, используемые для повышения локальности данных. Эти преобразования позволяют сократить количество перезагрузок кэша при выполнении цикла, что приводит к существенному увеличению скорости выполнения программы.
  Более того, будет показано, что обеспечение локальности данных является необходимым условием эффективного распараллеливания цикла.