Оглавление

[**Оптимизация циклов (loop optimization)** 2](#_Toc534479425)

[**Развертка (loop unrolling)** 2](#_Toc534479426)

[**Объединение циклов (loop fusion)** 3](#_Toc534479427)

[**Разрезание циклов (loop distribution)** 4](#_Toc534479428)

[**Перестановка циклов (loop interchange)** 5](#_Toc534479429)

[**Де-факторизация циклов (Loop De-factorization)** 6](#_Toc534479430)

[**Разбивка на блоки (Loop tiling)** 7](#_Toc534479431)

[**Инверсия (Loop inversion)** 8](#_Toc534479432)

[**Циклическое преобразование кода (Loop-invariant code motion)** 9](#_Toc534479433)

[**Loop unswitching** 10](#_Toc534479434)

[**Ветвления** 11](#_Toc534479435)

[**Индексные массивы** 14](#_Toc534479436)

[**I способ.** 14](#_Toc534479437)

[**II способ.** 15](#_Toc534479438)

[**III способ.** 15](#_Toc534479439)

[**Оптимизация массивов** 17](#_Toc534479440)

[**Устройство кэш памяти** 19](#_Toc534479441)

[**Три различные стратегии по заполнению кэша.** 22](#_Toc534479442)

[**Отображение RAM на кеш-память** 23](#_Toc534479443)

# **Оптимизация циклов (loop optimization)**

## **Развертка (loop unrolling)**

**«Когда используем?»:** данный вид оптимизации используется при малом теле цикла (с известной длиной), вместо цикла с n-итерациями, компилятор будет производить код, который просто повторяется n раз.

**«Зачем?»:** может повысить производительность, так как устраняет любые инструкции перехода, что является целью многих оптимизаций.

**Недостатки:** увеличенный размер кода.

**Пример:**

**До оптимизации:**

int i;

for ( i = 1; i < n; i++)

{

a[i] = (i % b[i]);

}

**После:**

int i;

for (i = 1; i < n - 3; i += 4)

{

a[i] = (i % b[i]);

a[i + 1] = ((i + 1) % b[i + 1]);

a[i + 2] = ((i + 2) % b[i + 2]);

a[i + 3] = ((i + 3) % b[i + 3]);

}

for (; i < n; i++)

{

a[i] = (i % b[i]);

}

## **Объединение циклов (loop fusion)**

**«Когда используем?»:** при возможности объединения смежных циклов в один

**«Зачем?»:** уменьшение дополнительных расходов на циклы, для параллельных систем помогают уменьшить запуск и, возможно, затраты на синхронизацию

**Недостатки:** не всегда выигрываете в производительности, так как тело цикла может «вывалиться» из кэша и в этом случае лучше применить разрезание циклов. Инициализировать два массива в отдельных циклах, иногда лучше, чем оба массива одновременно в одном цикле.

**Пример:**

**До:**

for(int i = 0; i < iN; i++){  
 a[i] = b[i] - 5;  
}  
for(int i = 0; i < iN-1; i++){  
 d[i] = e[i] \* 3;  
}

**После:**

for(int i = 0; i < iN-1; i++){  
 a[i] = b[i] - 5;  
 d[i] = e[i] \* 3;  
}  
a[iN-1] = b[iN-1] - 5;

## **Разрезание циклов (loop distribution)**

**«Когда используем?»:** большое тело программы не помещается в кэш, начало вытесняет конец и наоборот, а также данные вытесняются в кэш, в худшей ситуации в оперативную память (ОП), что существенно замедляет программу.

**«Зачем?»:** удалить последовательные операторы в отдельный цикл и собрать параллелизуемые операторы в другой цикл.

**Недостатки:** применение может мешать зависимости по данным

**Пример:**

**До:**

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

**После:**

/\* L1: parallel loop \*/

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

/\* L2: sequential loop \*/

for (i=0; i < n; i++) {

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

}

## **Перестановка циклов (loop interchange)**

**Важно: (но не во всех случаях)**

* Внешним был цикл наиболее выгодный для параллелизации.
* Самым внутренним должен быть более длинный цикл.
* Самым внутренним должен быть цикл, оперирующий элементами, расположенными подряд в оперативной памяти.

**«Когда используем?»:** когда неправильный порядок вложения циклов (см. пункт выше), либо игра c кэш памятью, либо регистры (лучше внутренним делать тот, который хорошо раскладывается по регистрам)

**«Зачем?»:** помочь компилятору распараллеливать код

**Недостатки:** если данные идут не последовательно, то оптимизация – проигрывает

**Пример:**

**До:**

for(int i = 0; i < iN; i++){  
for(int j = 0; j < jN; j++){  
for(int k = 0; k < kN; k++){  
 c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

**После:**

for(int i = 0; i < iN; i++){   
for(int k = 0; k < kN; k++){  
for(int j = 0; j < jN; j++){   
  c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

Конечный цикл можно объединить.

## **Де-факторизация циклов (Loop De-factorization)**

**«Когда используем?»:** когда можем преобразовать математические формулы, которые будут более «дешёвыми» для вычисления

**Математические справочники:**

**1.** Бронштейн И. Н. и др. Справочник по математике.

**2.** Корн Г., Корн Т. Справочник по математике.

**3.** Абрамовиц М., Стиган И. Справочник по специальным функциям с формулами, графиками и таблицами

**«Зачем?»:** выигрываем в быстроте, к примеру, умножение ~ 4 такта, а деление ~ 1.5 десятка тактов

**Пример:**

**До:**

cos (x+ y)

sin (x)

a/x   
b/x  
c/x

**После:**

cos(x)\*cos(y) – sin(x)\*sin(y)

eix-e-ix/2\*i

tmp = 1/x  
a\*tmp  
b\*tmp  
c\*tmp

## **Разбивка на блоки (Loop tiling)**

**«Когда используем?»:** если тело цикла сложное, то можно применить эту оптимизацию и разбить цикл на более маленькие части

**«Зачем?»:** разбиение пространства итерирования цикла приводит к разбиению массива на меньшие блоки, которые помещаются в кэш, что приводит к улучшению использования кэша, снижению количества промахов

**Пример:**

**До:**

for (i = 0; i < N; i++) for (j = 0; j < N; j++)

c[i] = c[i] + a[i, j] \* b[j];

**После:**

for (i = 0; i < N; i += 2)

for (j = 0; j < N; j += 2)

for (ii = i; ii < min(i + 2, N); ii++)

for (jj = j; jj < min(j + 2,N); jj++)

c[ii] = c[ii] + a[ii, jj] \* b[ii];

## **Циклическое преобразование кода (Loop-invariant code motion)**

**«Когда используем?»:** когда можно вынести одноразовые вычисления за цикл, чтобы не вычислять их несколько раз

**«Зачем?»:** значительно повышает производительность

**Пример:**

**До:**

for (int i = 0; i < n; ++i) {

x = y + z;

a[i] = 6 \* i + x \* x;

}

**После:**

x = y + z;

t1 = x \* x;

for (int i = 0; i < n; ++i) {

a[i] = 6 \* i + t1;

}

## **Loop unswitching**

**«Когда используем?»:** когда условие внутри тела цикла мешает его распараллеливанию

**«Зачем?»:** на современных процессорах ради скорости, так как они быстро работают на векторах

**Пример:**

**До:**

int i, w, x[1000], y[1000];

for (i = 0; i < 1000; i++) {

x[i] += y[i];

if (w)

y[i] = 0;

}

**После:**

int i, w, x[1000], y[1000];

if (w) {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

y[i] = 0;

}

} else {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

} }

# 

**Ветвления**

В состав современных процессоров входит модуль предсказания ветвлений – branch predictor. Всякий раз, когда код имеет ветвь (например, if-структуру) или если мы находимся внутри цикла, микропроцессор не знает заранее, какая из двух ветвей нужна для подачи в  pipeline. Branch predictor нужен для того, чтобы предсказать по какой ветке пойдет if – останемся ли мы внутри тела цикла или выйдем из него.

Сам процессор работает быстрее, чем оперативная память, из которой он берет данные. Поэтому желательно обращаться к памяти таким образом, чтобы данные заведомо находились либо в регистрах процессора, либо в кэш памяти. В таком случае, процессор может эти данные подтянуть, если будет знать, куда пойдут вычисления.

Вычисления развиваются линейно, поэтому предсказать, что понадобится через некоторое количество тактов процессора достаточно легко, если в этом коде нет ветвлений.

Например, конструкция ветвления с оператором if – самый плохой случай, т.к. оператор if предусматривает различные длинные последовательности вычислительных команд.

Пример конструкции ветвления с оператором  if:

**if** (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} **else** { *// иначе*

cout << "Это число больше либо равно 10." << endl;

}

Усовершенствуем программу следующим образом:

**if** (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} **else** **if** (num == 10) {

cout << "Это число равно 10." << endl;

} **else** { *// иначе*

cout << "Это число больше 10." << endl;

}

`Рассмотрим короткие ветвления, которые в большинстве случаев имеют преимущества над длинными. Их разумно выполнять в виде конструкции – с оператором “?:” ( x = a>b ? y : z; ).

Пример:

if (a > 0)

{x = 1, y = 2, z++ }

else {q = 3 z = 4 y = 5}

Можно переписать с помощью одной тернарной операции.

Для начала нужно найти общие присваемые части в обоих случаях – это z и y. В z выполняются разные операции: z++ - инкрементируется; z = 4 записывается. А y в обоих случаях просто записывается, что довольно удобно.

y = a > 0 ? { x = 1, z++, 2} – непрерывная тернарная операция

y > 0 ?( x = 1, z++, 2) : (q = 3 z = 4 y = 5)

В данном методе исключается предсказатель ветвления – просто выполняется логическое выражение. Данный метод выигрывает в скорости выполнения.

y = a > 0 ? 2:5;

x = a > 0 ? 1:x;

z = a > 0 ? z++:4;

q = a > 0 ? q:3

Другим методом оптимизации ветвлений является устранение ветвления, а именно замена на выбор из таблицы. Вместо условного выражения, в котором оба случая являются константами, используется справочная таблица. Если есть оператор, подобный следующему, где c и d представляют константные выражения, а b представляет логическое выражение:

a = b ? c : d

Это эквивалентно следующему:

if (b) a = c;

else a = d;

Заменим следующим эквивалентным и более быстрым кодом:

static const type lookup\_table[] = { d, c };

a = lookup\_table[b];

Условное выражение компилируется в ветвь. Если такая ветвь не очень хорошо предсказана, это займет больше времени, чем таблица соответствия.

a = b1 ? c : b2 ? d : b3 ? e : f;

Это эквивалентно следующему:

if (b1) a = c;

else if (b2) a = d;

else if (b3) a = e;

else a = f;

Заменим следующим эквивалентным кодом:

static constant type lookup\_table [] = { f, e, d, d, c, c, c, c };

a = lookup\_table [b1 \* 4 + b2 \* 2 + b3];

Оператор switch. switch, всегда можно заменить на константный массив.  
  
Под последовательными значениями представляется:

switch (a) {

case 0:

case 1:

case 2:

…

}

Безусловно switch работает быстрее многоуровневых if else, но все-равно это условие. Для повышения производительности всегда можно воспользоваться константным массивом.  
Пример:

int a, b = 0;

/\*switch (a) {

case 0: b\* = 10;

case 2: b\* = 20;

case 3: b\* = 30;

…

}\*/

const int array[3] = {10,0,20,30};

b \*= array[a];

Самый основной (неоптимизированный) способ составления оператора switch - это трактовать его как цепочку if … else if … операторов. Обычный способ, которым компиляторы оптимизируют переключатель, - это преобразовать его в таблицу переходов, которая может выглядеть примерно так:

if (condition1) goto label1;

if (condition2) goto label2;

if (condition3) goto label3;

else goto default;

label1:

<<<code from first `case statement`>>>

goto end;

label2:

<<<code from first `case statement`>>>

goto end;

label3:

<<<code from first `case statement`>>>

goto end;

default:

<<<code from `default` case>>>

goto end;

end:

Одна из причин, по которой этот метод работает быстрее, заключается в том, что код внутри условных выражений меньше (поэтому при неправильном прогнозировании условного кэша будет меньше штраф). Кроме того, «провальный» случай становится более тривиальным для реализации (компилятор пропускает goto end оператор).

Компиляторы могут дополнительно оптимизировать таблицу переходов путем создания массива указателей (на места, отмеченные метками) и использовать значение, которое вы включаете, в качестве индекса в этом массиве. Это исключило бы почти все условные выражения из кода (за исключением того, что было необходимо для проверки того, соответствует ли значение, которое вы включаете, одному из ваших случаев или нет).

Важно отметить, что вложенные таблицы переходов трудно генерировать, и некоторые компиляторы отказываются даже пытаться их создать. По этой причине стоит избегать вложения switch внутрь другого, switch если важен максимально оптимизированный код.

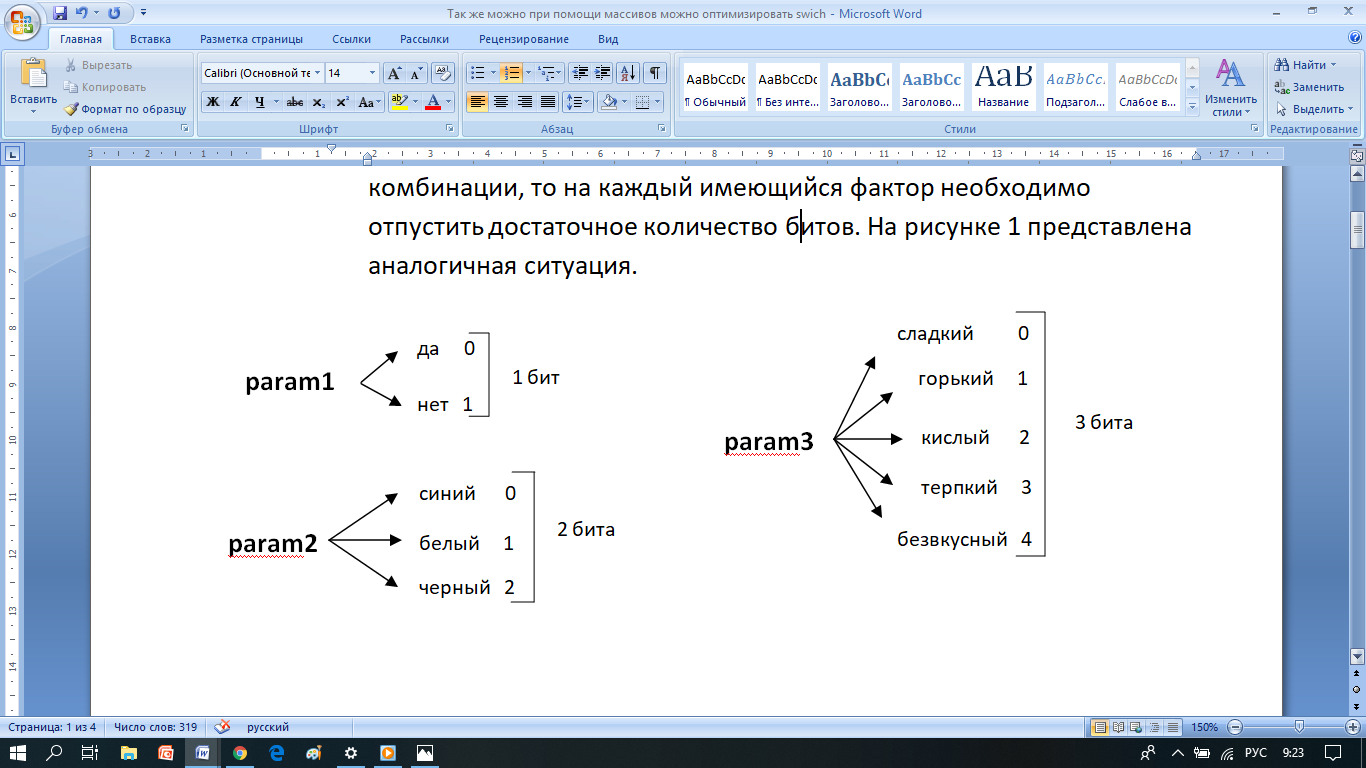
# **Индексные массивы**

Постановка задачи: имеется много разных параметров (**param**), от которых зависит решение. Все параметры независимы. Каждый из них принимает конечное количество вариантов.

Поставленную задачу можно решить несколькими способами:

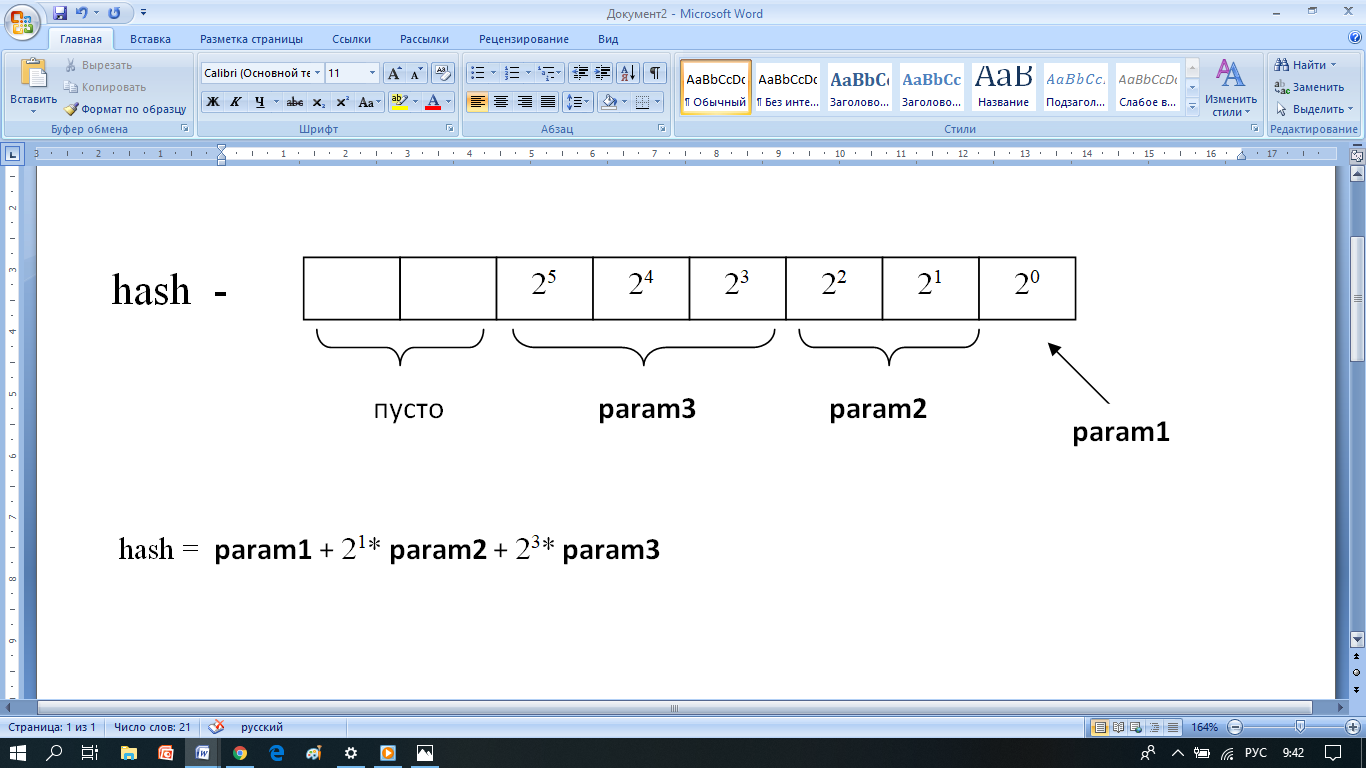
I способ. *Кодирование всех возможных вариантов.*

1. Так как параметры независимые и встречаются все возможные комбинации, то на каждый имеющийся параметр необходимо отпустить достаточное (но не более того) количество битов. На рисунке 1 представлен пример подобной ситуации.



*Рисунок 1. Кодирование факторов*

1. Из этих коротких целых чисел, каждое из которых содержит маленькое количество битов, путем сдвигов и монтажа собирается одно число, которое в итоге попадает на выражение switch. Проиллюстрировано это на рисунке 2.



*Рисунок 2. Формирование hash*

I I способ. *Кодирование только возможных сочетаний.*

Этот способ применятся тогда, когда не возможности закодировать все варианты, так как некоторые из них просто невозможны. В этом случае hash образуется следующим образом:

1. логическое выражение
2. a = a0 + (сложное логическое выражение 1)\* a1

b = b0 + (сложное логическое выражение 2)\* b1

c = c0 + (сложное логическое выражение 3)\* c1

Например, если положить a0 = b0 = c0 = 0, а a1 = b1 = c1 = 1, то

hash = a+2\*b+4\*c

I I I способ. *Индексные массивы (index arrays/ Look Up Table).*

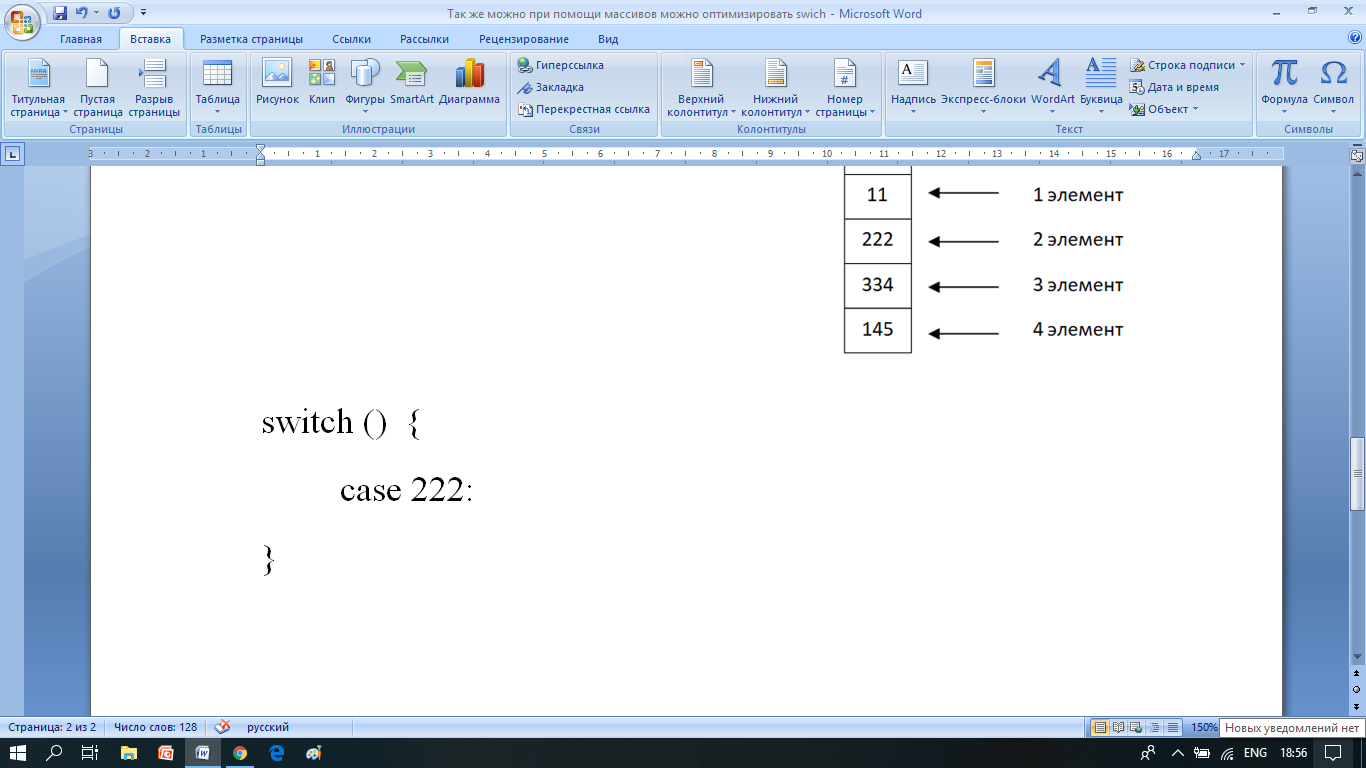
1. Создадим вспомогательный массив. Исходные ситуации описываются числами, которые идут на вход switch. На рисунке 3 представлена визуализация индексного массива.



*Рисунок.3. Индексный массив*

Это необходимо для того, чтобы непрерывно идущий числовой ряд (элемент 0, 1, …,4) превратить в необходимые для switch значения, которые могут идти в различном порядке.

1. Выполняем промежуточную перекодировку (рисунок 4).



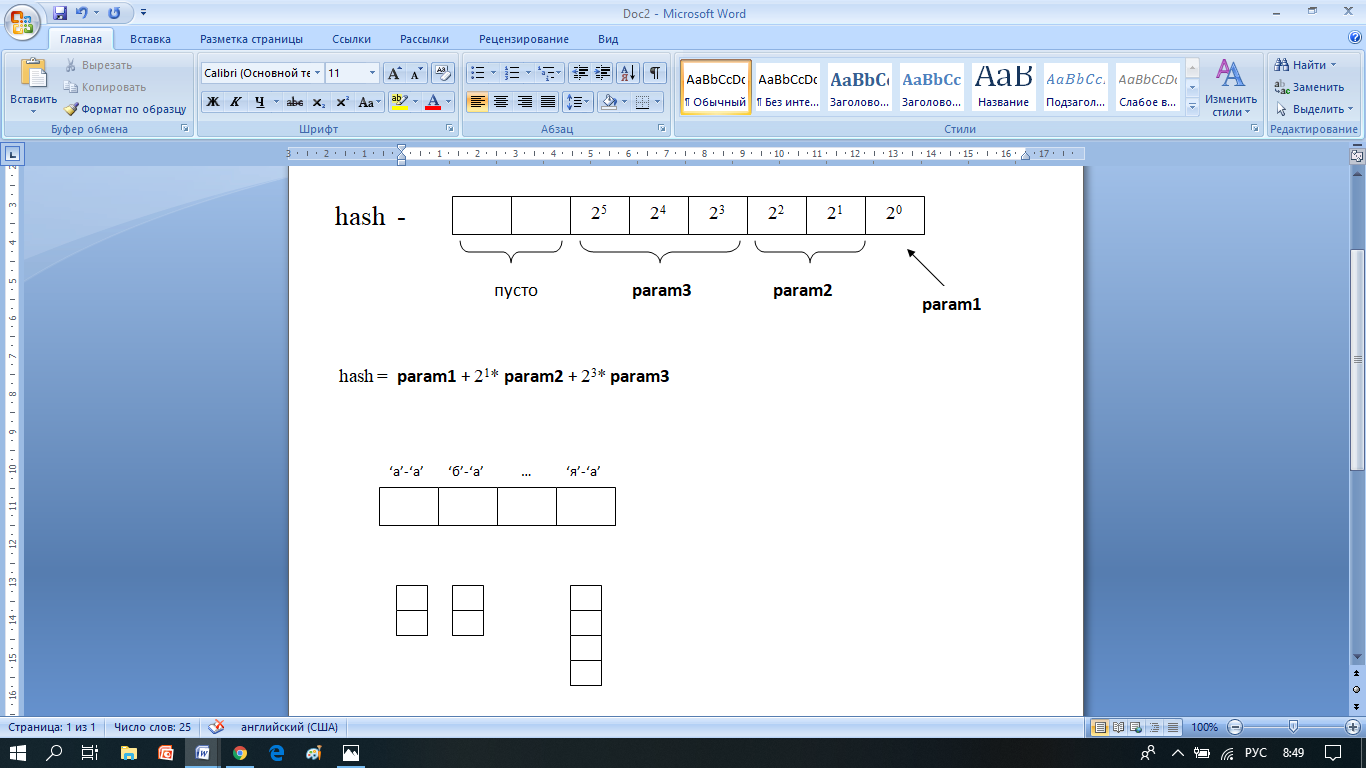
*Рисунок 4. Использование индексного массива*

Рассмотрим на примере задачи как можно применить данный метод на практике.

*Задача*. Есть определенное правило, по которому осуществляется транслитерация, т.е. на вход поступает русский текст, известно правило, по которому всякой русской букве или буквосочетание можно поставить в соответствие латинскую букву или буквосочетание. Необходимо превратить входной тест в тест в соответствии с заданным правилом. (Например, с s, ш sh).

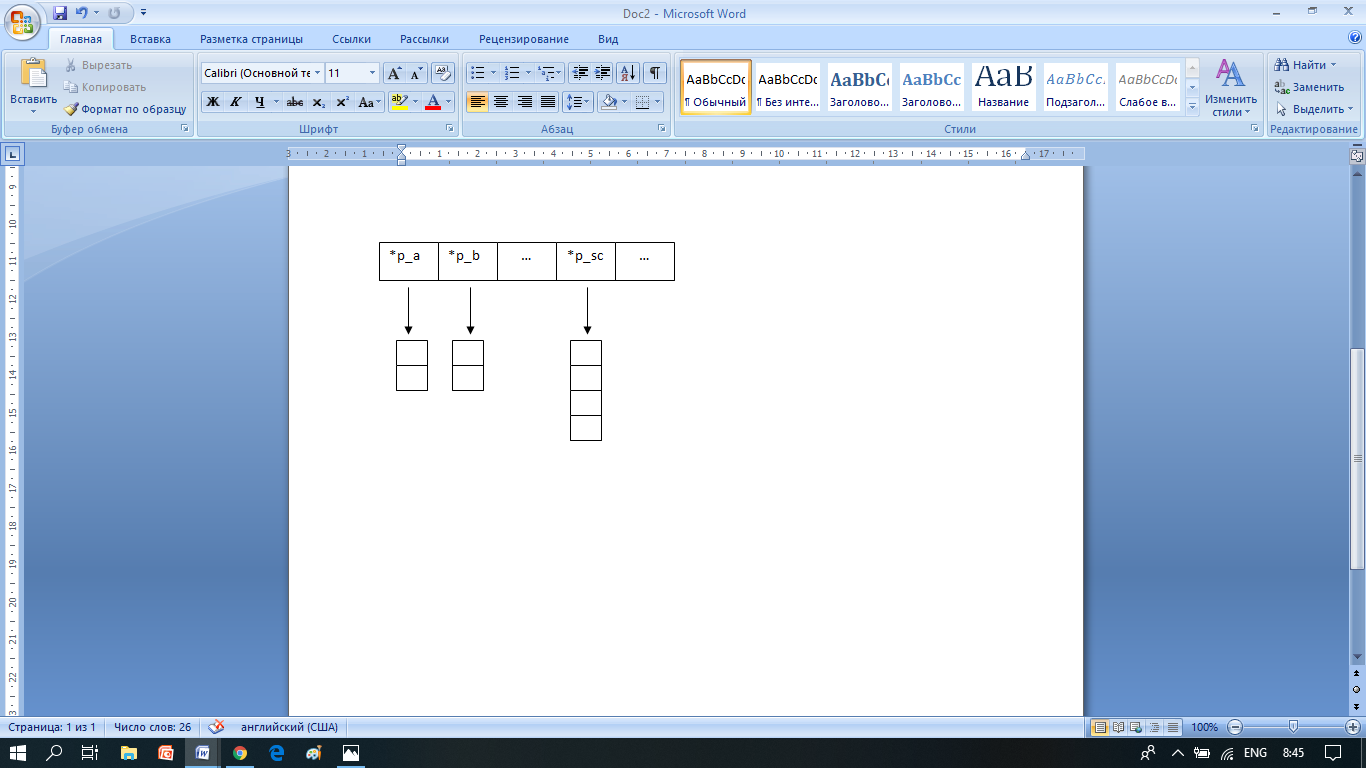
Нужно максимально оптимизировать транслитерацию.

1. Формируем вспомогательный массив, таким образом, чтобы индексы массива получаются посредством вычитания из кода буквы кода буквы ‘а’. Например, 0 индекс формируется как ‘а’-‘а’, а 1 – ‘б’-‘а’(рисунок 5).



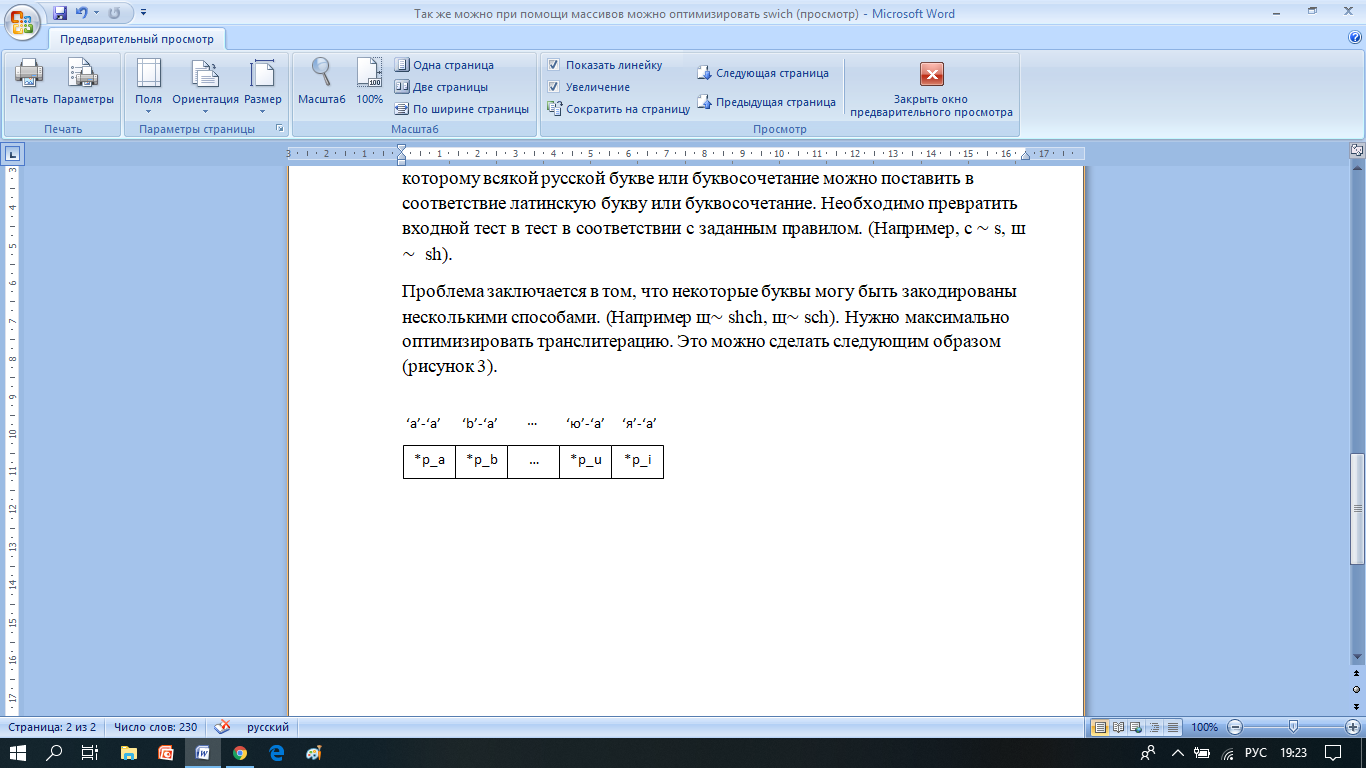
*Рисунок 5. Индексы массива*

1. В каждую ячейку массива кладем указатель на массив, где храниться транслитерация отдельно взятой буквы.
2. Указатели на массивы с транслитерацией буквы могут быть разной длины, потому что некоторый буквы транслитерируются несколькими символами. Например, ‘щ’ ≈ sch (рисунок 6).



*Рисунок 6. Указатели на массивы*

1. Результат изображен на рисунку 7.



*Рисунок 7. Использование индексного массива для транслитерации*

Пример 1.

char \*p\_a=PointerToA;  
...  
char \*p\_zh=PointerToZh;

…  
switch (character){  
 case 'а'-'а':  
 \*p\_a;  
 break;  
 ...  
 case 'ж'-'а':  
 \*p\_zh;  
 break;  
 …   
}

# **Оптимизация массивов**

Массив это структура данных, представленная в виде ячеек определенного типа, объединенных одним именем. Массивы используются, если необходимо обработать большое количество данных. Его имя является постоянным указателем, который указывает на первый элемент массива. Это значит, что можно использовать указатели и их арифметику.

Обращение к элементам массива выглядеть, как показано в примере 1.

Пример 1.

for (int i=0; i<n; i++)

array [i] = n\_some\_value;

Вместо приведенного выше кода эффективнее сделать, как показано в примере 2.

Пример 2.

int\* arrn=array +n;

for (int \* ptr\_int=array; ptr\_int<arrn; ptr\_int ++)

\* ptr\_int= n\_some\_value;

Причина этого в операциях с указателями. В примере 2 есть указатель на тип int, который берет адрес из имени массива array и увеличивает его для одного элемента.

В примере 3 рассмотрена очевидная обработка двумерного массива. ( В данном случает его обнуление).

Пример 3

for (int i=0; i<n\_rows; i++) {

for (int j=0; j<n\_cols; j++) {

array[i][j]=0;

}

}

Использование указателей сделает код из примера 3 более эффективным.

Пример 4

int \*\*p = array;

int \*\*end = array \* n\_rows +n\_cols;

for (p = array; p < end; p++) {

\*\*p = 0;

}

Используя оптимизация из примера 4 можно ускорить обработку массива приблизительно в 10 раз. Обработка массива происходит быстрее с указателями. Вместо того, чтобы вычислять позицию каждого элемента с помощью индексов, указатель перемещался по массиву, увеличивая на одну целую позицию после каждой итерации до конца – адреса, который был вычислен один раз, заранее.

Конечно, хороший оптимизирующий компилятор упростил был код примера 4.

В примере 5 требуется посчитать сумму элементов двумерного массива

Пример 5.

int a[n][m];

int sum =0;

for (int i=0; i<n;i++){

for (int j=0; j<m;j++ )

sum+=a[i][j];

}

Время выполнения фрагмента кода из примера 5 может сильно увеличить

при достаточно больших n и m, потому что на каждой итерации цикла при

определении адреса [i][j] – го элемента компилятор использует много

вычислений.

Зная о том, что элементы массива расположены в памяти последовательно,

можно оптимизировать пример 5, введя вспомогательный указатель.

Пример 6.

int \*p= &a[0][0];

float size=sizeof(a)/sizeof(int);

for (int i=0; i<size;i++){

sum+=\*p;

p++;

}

Так же можно использовать эквивалентное объявление указателя \*p= a[0].

Пример 7.

int \*p; //указатель на массив элементов

x=\*(p++);

у=\*(++p);

В примере 7 первое присваивание значительно эффективней, чем второе. В первом случае будет осуществляться разыменование указателя и его инкремента параллельно, а во втором – последовательно.

Еще один совет по оптимизации работы с двумерными массивами.

Количество столбцов двумерного массива желательно должно быть равно

степени двойки при небольших типах данных. Это ускорит работу с массивом, поскольку это выравнивает указатели на первые элементы каждой строки, что ускоряет доступ к элементам.

Пример 8.

double array [7][8]; // 8 =23

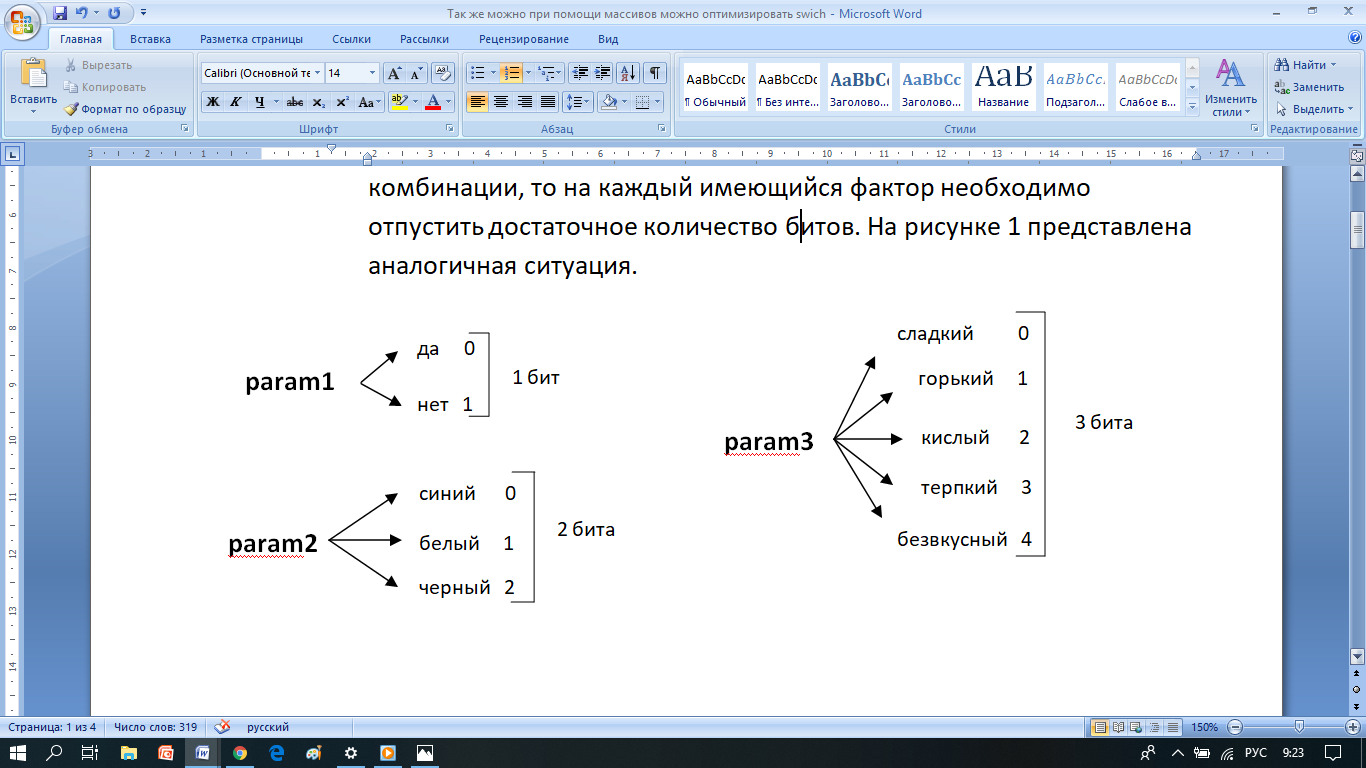
**Оптимизация switch.**

Постановка задачи: Имеется много разных параметров (**param**), от которых зависит решение. Все параметры независимы. Каждый из них принимает конечное количество вариантов.

Поставленную задачу можно решить несколькими способами:

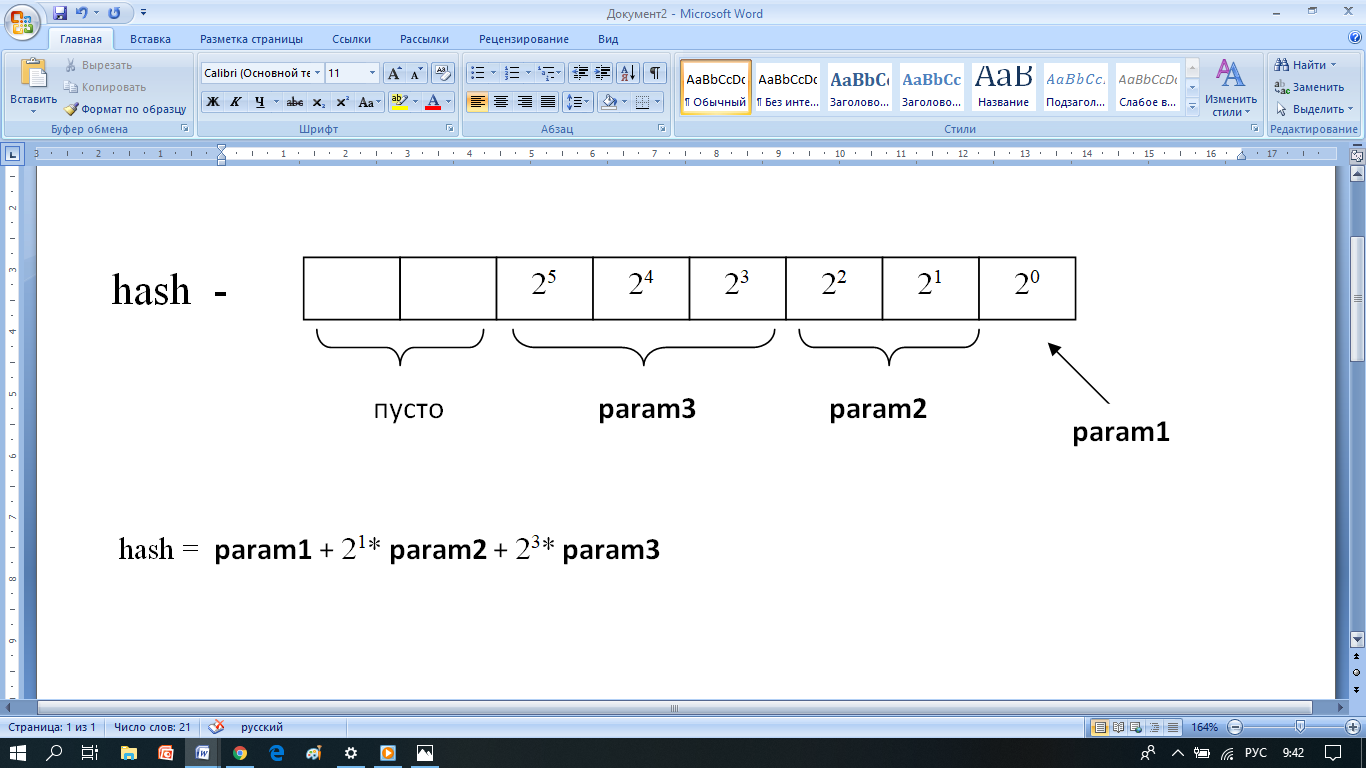
I способ. *Кодирование всех возможных вариантов.*

1. Так как параметры независимые и встречаются все возможные комбинации, то на каждый имеющийся параметр необходимо отпустить достаточное (но не более того) количество битов. На рисунке 1 представлен пример подобной ситуации.



*Рисунок 1. Кодирование факторов*

1. Из этих коротких целых чисел, каждое из которых содержит маленькое количество битов, путем сдвигов и монтажа собирается одно число, которое в итоге попадает на выражение switch. Проиллюстрировано это на рисунке 2.



*Рисунок 2. Формирование hash*

I I способ. *Кодирование только возможных сочетаний.*

Этот способ применятся тогда, когда не возможности закодировать все варианты, так как некоторые из них просто невозможны. В этом случае hash образуется следующим образом:

1. логическое выражение
2. a = a0 + (сложное логическое выражение 1)\* a1

b = b0 + (сложное логическое выражение 2)\* b1

c = c0 + (сложное логическое выражение 3)\* c1

Например, если положить a0 = b0 = c0 = 0, а a1 = b1 = c1 = 1, то

hash = a+2\*b+4\*c

I I I способ. *Индексные массивы (index arrays/ Look Up Table).*

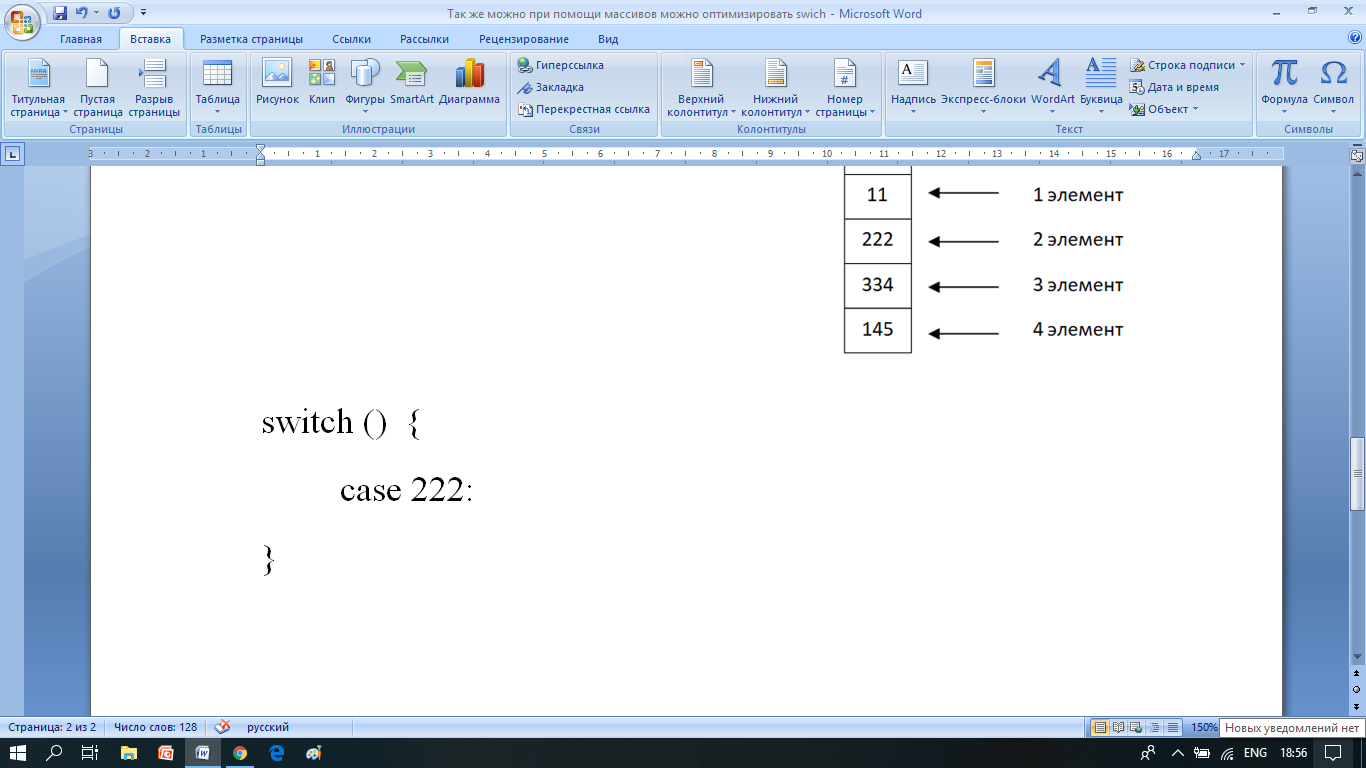
1. Создадим вспомогательный массив. Исходные ситуации описываются числами, которые идут на вход switch. На рисунке 3 представлена визуализация индексного массива.



*Рисунок.3. Индексный массив*

Это необходимо для того, чтобы непрерывно идущий числовой ряд (элемент 0, 1, …,4) превратить в необходимые для switch значения, которые могут идти в различном порядке.

1. Выполняем промежуточную перекодировку (рисунок 4).



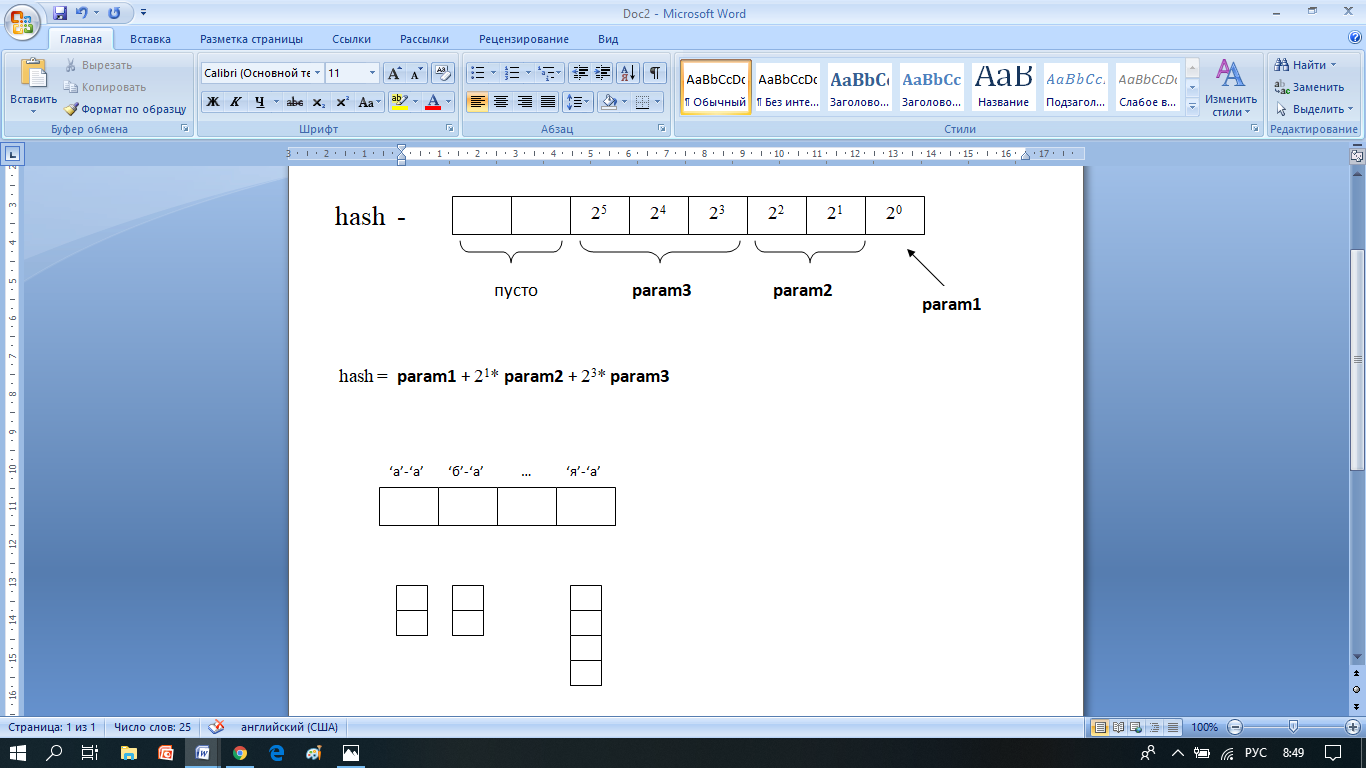
*Рисунок 4. Использование индексного массива*

Рассмотрим на примере задачи как можно применить данный метод на практике.

*Задача*. Есть определенное правило, по которому осуществляется транслитерация, т.е. на вход поступает русский текст, известно правило, по которому всякой русской букве или буквосочетание можно поставить в соответствие латинскую букву или буквосочетание. Необходимо превратить входной тест в тест в соответствии с заданным правилом. (Например, с s, ш sh).

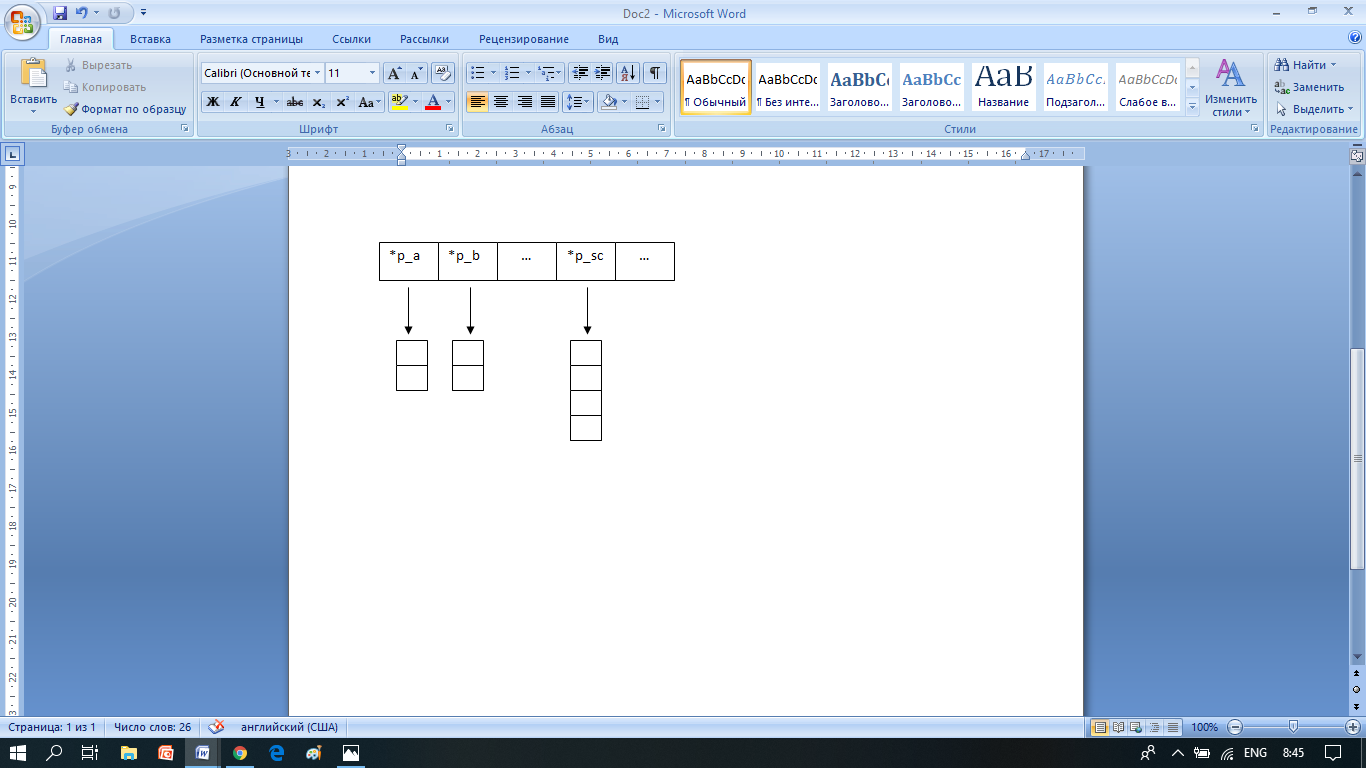
Нужно максимально оптимизировать транслитерацию.

1. Формируем вспомогательный массив, таким образом, чтобы индексы массива получаются посредством вычитания из кода буквы кода буквы ‘а’. Например, 0 индекс формируется как ‘а’-‘а’, а 1 – ‘б’-‘а’(рисунок 5).



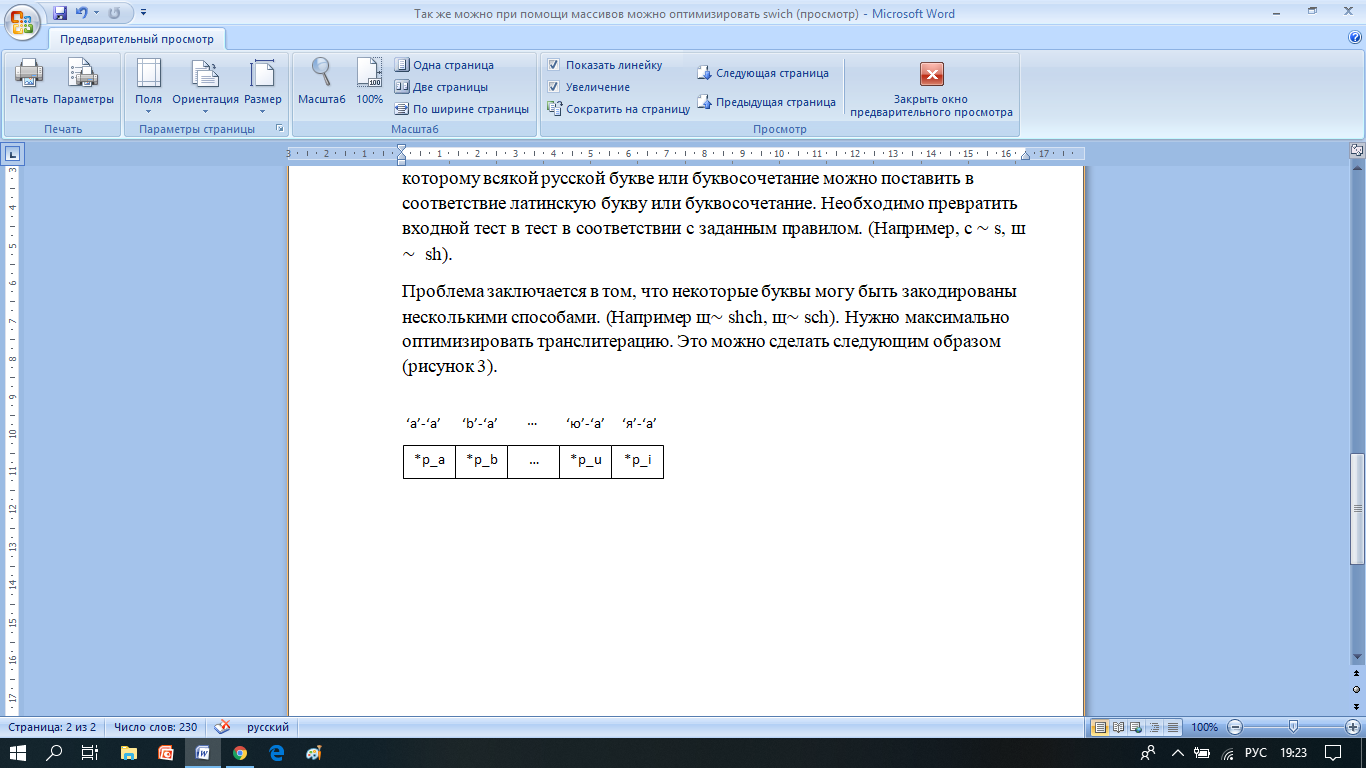
*Рисунок 5. Индексы массива*

1. В каждую ячейку массива кладем указатель на массив, где храниться транслитерация отдельно взятой буквы.
2. Указатели на массивы с транслитерацией буквы могут быть разной длины, потому что некоторый буквы транслитерируются несколькими символами. Например, ‘щ’ ≈ sch (рисунок 6).



*Рисунок 6. Указатели на массивы*

1. Результат изображен на рисунку 7.



*Рисунок 7. Использование индексного массива для транслитерации*

Пример 1.

char \*p\_a=PointerToA;  
...  
char \*p\_zh=PointerToZh;

…  
switch (character){  
 case 'а'-'а':  
 \*p\_a;  
 break;  
 ...  
 case 'ж'-'а':  
 \*p\_zh;  
 break;  
 …   
}

# **Устройство кэш памяти**

* **Кэш память** – сверхбыстрая память, расположенная ближе всех к процессору. Обычно она быстрее оперативной памяти примерно в 2 или более раз. Хранит в себе копии часто используемых данных из обычной памяти.
* Кэш память обычно представляет собой трехуровневую систему:

Первый уровень(L1) – самая быстрая память, но самая маленькая. Нет задержек, бывает двух типов: пространство инструкций и пространство данных. На каждое ядро выделено по каждому из этих типов.

Второй уровень(L2) – немного медленней L1, но объем ее при

этом в десятки раз больше чем у L1. Хранит в себе менее часто используемые данные, по сравнению с L1, нет разделения на инструкции и данные. Для каждого ядра процессора устанавливается свой кэш L2, то есть у одного ядра нет доступа к кэшу другого.

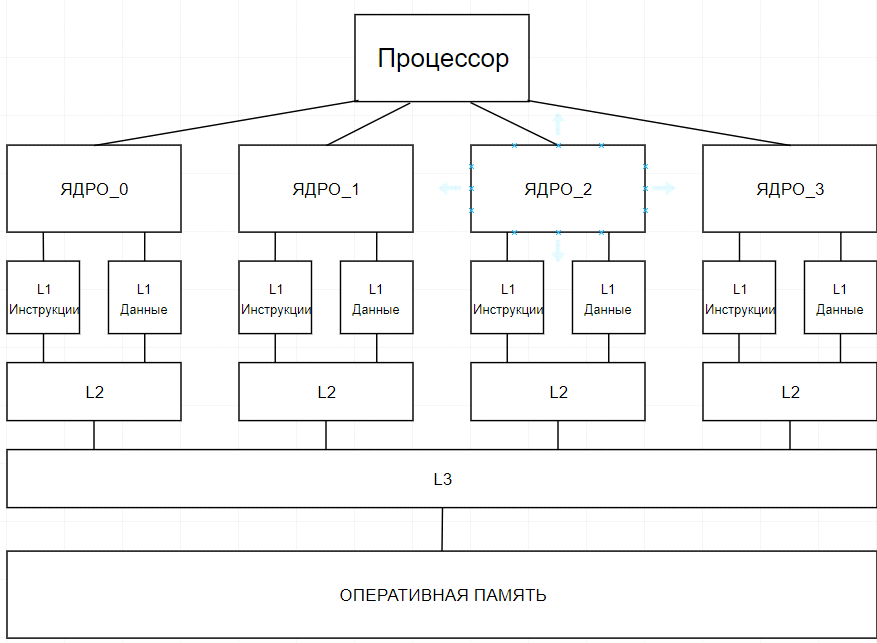
 Третий уровень(L3) – медленнее, чем L2, но по объему является самой большой. Размер может достигать нескольких мегабайт. Является общей для всех ядер процессора.

Рис1.Архитектура кэш памяти процессора

* Контроллер кэш-памяти.

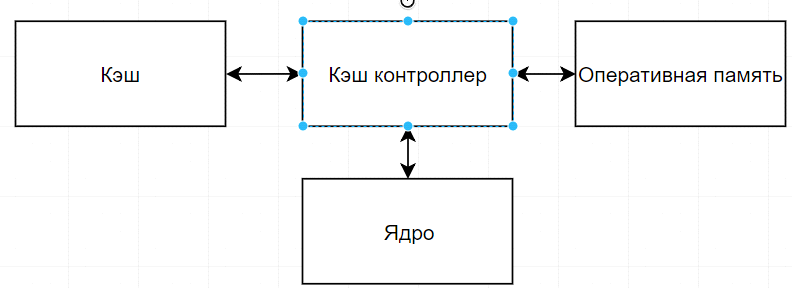
Это устройство, передающее данные между кэш памятью, оперативной памятью и процессором. Когда ядро обращается за данными, контроллер проверяет их наличие в кэш памяти. Если данные на месте, то происходит немедленная передача данных ядру, в противном случае(*промах кэша-cache miss*) придется ожидать поступления данных из оперативной памяти. Задача контроллера – сделать так, чтобы промахов кэша было как можно меньше.

Рис2. Взаимодействие контроллера с процессором и памятью

4.**Организация кэш памяти**

Кэш-память - это набор кэш-строк. Каждая кэш-строка хранит блок данных определенного размера и дополнительную информацию(например, тэг). Под размером кэш-строки понимают обычно размер блока данных, который в ней хранится.

Для задания адреса каждого байта памяти требуется x-байтный адрес. Предположим, что наш кэш работает на уровне отдельных байтов, то есть может сохранять в качестве элемента байт оперативной памяти. Тогда каждая единица кэша должна сохранять байт данных оперативной памяти и его x-байтный адрес в оперативной памяти. Получается строка, которая называется кэш­строкой. Адрес сохраняемого байта – тэг(tag). При чтении данных из кэша процессор формирует адрес, который сравнивается кэш-контроллером с тегом кэш­строки. В случае совпадения кэш-контроллер выдает требуемый байт данных, если же совпадения адреса с тегом нет (кэш­промах) — производится обращение к оперативной памяти, что не есть хорошо, так как ядро. Примечание: размер адреса – х, зависит от разрядности системы.

* Подробнее о тэгах, их оптимизации.

Тэги очень дорого обходятся для системы по следующим причинам:

* логика сравнения тегов большая (операция XOR занимает лишний бит)
* логика сравнения тегов медленная
* слишком большие затраты памяти на хранение тэгов

Например, в полностью ассоциативном отображении RAM на кэш запись тега в кэш обходится в 30 битов, а запись данных к этому тэгу – 32, то есть 48% памяти кэша расходуется на хранение тэгов!

4.1. Уменьшение затрат на тэг: Больше данных на один тэг

Сделать кэш-линии больше: увеличить каждую строку в полностью ассоциативном кэше.

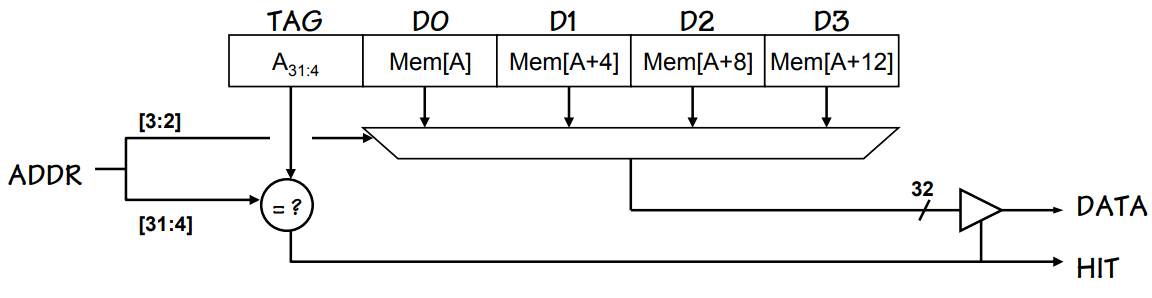


Рис.4. Увеличение кэш-линии

* Блоки из двоичных слов
* всегда читает/записывает двоичный слово-блок из/в память
* использует пространственную локализацию: соседние слова в блоке, вероятно, доступны
* стоимость: присутствуют некоторые наборы недоступных слов
* большой профит, если путь к памяти широк или последовательный доступ быстрый

В итоге получается, что на хранение тэгов необходимо 32 - 2 = 30 битов, а на хранение блока данных 4\*32 бита. Отсюда следует, что всего 18% от всей кэш памяти использовано на хранение тегов.

4.2. Соотношение промахов кэша(cache miss) к размеру блоков

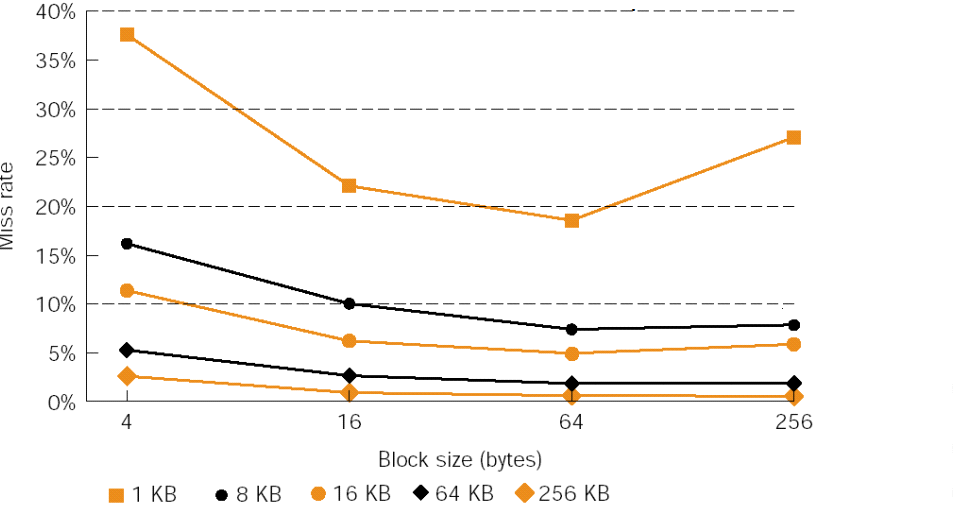


Рис.5 Соотношение промахов к размеру блоков

Если блоки данных сделать очень большими, то количество промахов

кэша в конечном итоге сильно увеличится и потому следует использовать

оптимальный размер блоков.

## 5. **Три различные стратегии по заполнению кэша.**

1) LRU (Least-recently used): заменяет объект, к которому дольше всего не обращались и предпочитает самые часто используемые данные. Лучшая производительность / высокая стоимость;

2) FIFO/LRR (first-in, first-out/least-recently replaced): заменяет самый старый элемент в кеше, предпочитает недавно загруженные элементы старым. Низкая производительность / экономичность;

3) Random: заменяет объекты случайным образом, нет фаворитов – все равны, использует псевдослучайный генератор, чтобы получить воспроизводимое поведение. Средняя производительность / самая низкая стоимость, избегает патологических последовательностей, но производительность может варьироваться;

## 6. **Отображение RAM на кеш-память**

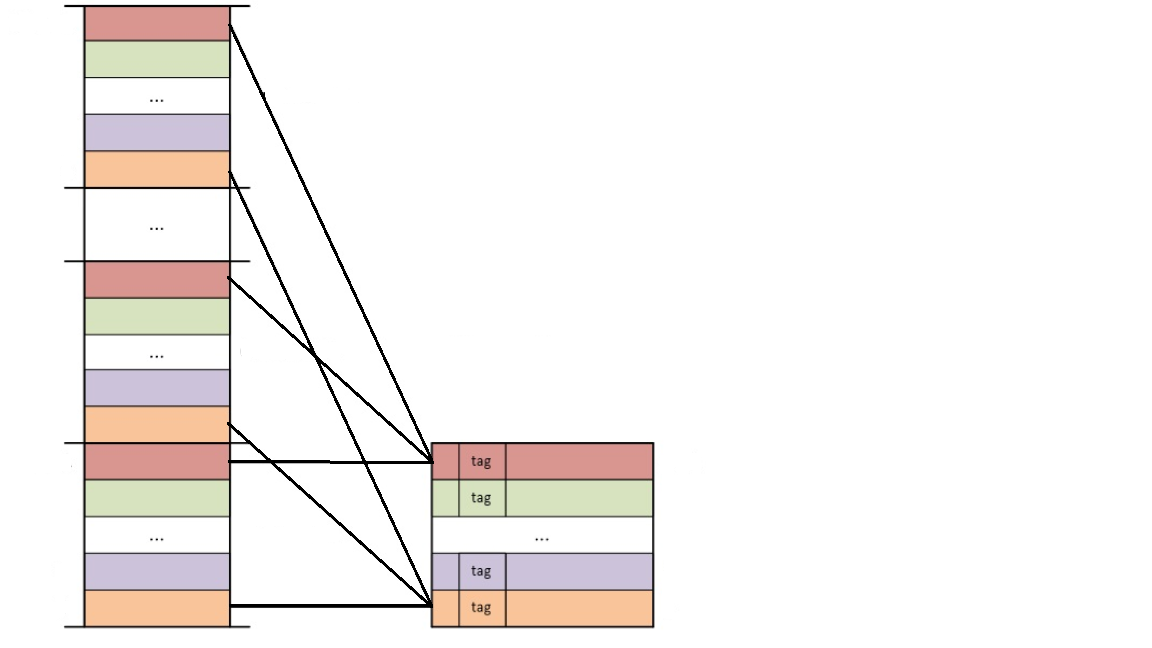
1)Direct-Mapped Cache (Прямое отображение): RAM делится на сегменты, размер каждого сегмента равен размеру кэша, а каждый сегмент в свою очередь делится на блоки, размер каждого блока равен размеру кэш строки. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на одну и ту же кэш строку. 

Рис.6 Устройство прямого отображения

Использовать стратегию для заполнения нет нужды, так как каждый блок ссылается на определенную кэш строку. Для хранения тэгов/данных используется SRAM.

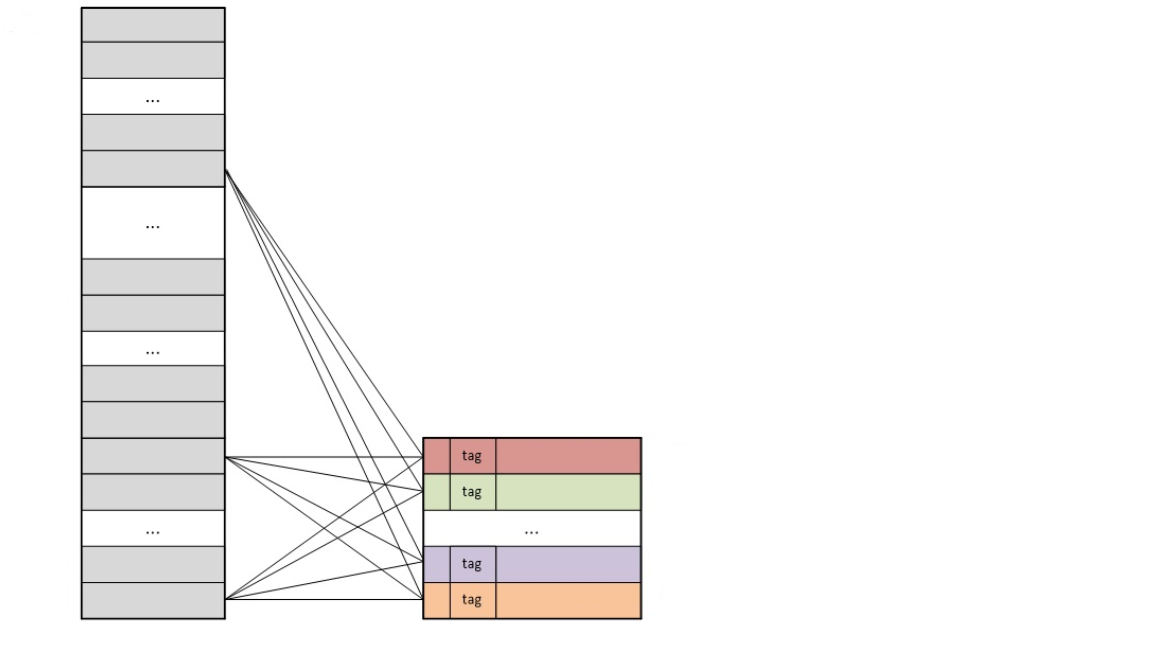
2)Fully Associative Cache (Полностью ассоциативное отображение): RAM делится на блоки, размер которых равен размеру кэш строки. Каждый блок имеет своё сравнивающее устройство и может отображаться в любую кэш строку. Устройство сравнивает адрес с тегом кэш строки, в случае успеха возвращает данные процессору. 

Рис.7 Устройство полностью ассоциативного отображения

Используются стратегии заполнения памяти для поиска кэш строки при добавлении новых данных. Для хранения тэгов/данных используются регистры.

3) N-Way Set-Associative Cache (Наборно-ассоциативное отображение): совмещает в себе два предыдущих способа отображения. RAM делится также, как и в прямом отображении, а сам кэш состоит из k кэшей, использующих прямое отображение. Кэш строки, имеющие одинаковые номера во всех каналах, образуют набор.

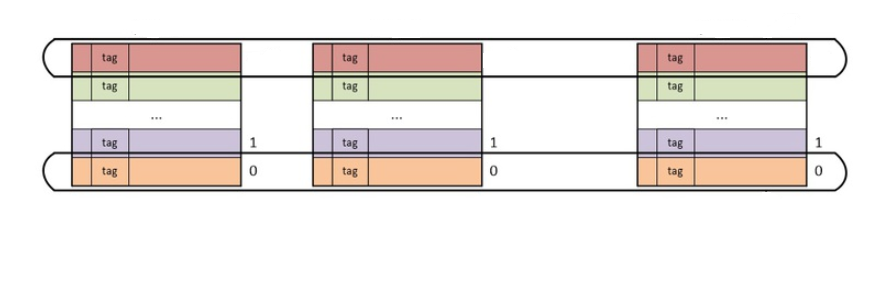


Рис.8 Набор в наборно-ассоциативном отображении

Набор представляет собой кэш с полностью ассоциативным отображением. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на один и тот же набор кэша.

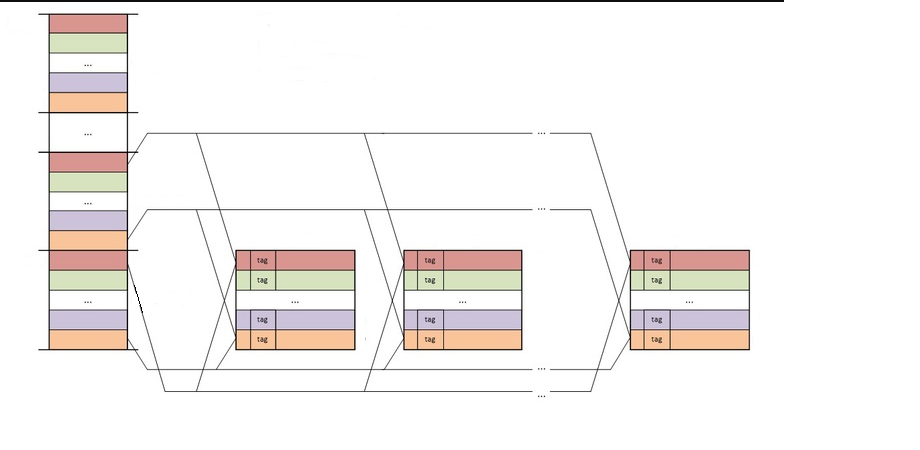


Рис.9 Устройство наборно-ассоциативного отображения

6.1 Соотношение промахов к ассоциативности

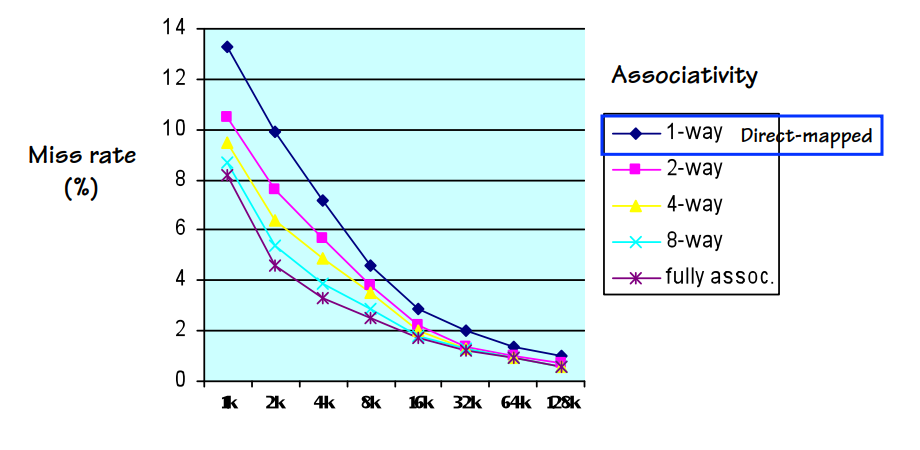


Рис.10 Соотношение промахов к ассоциативности

Видим, что более ассоциативное отображение способствует меньшее количество промахов.