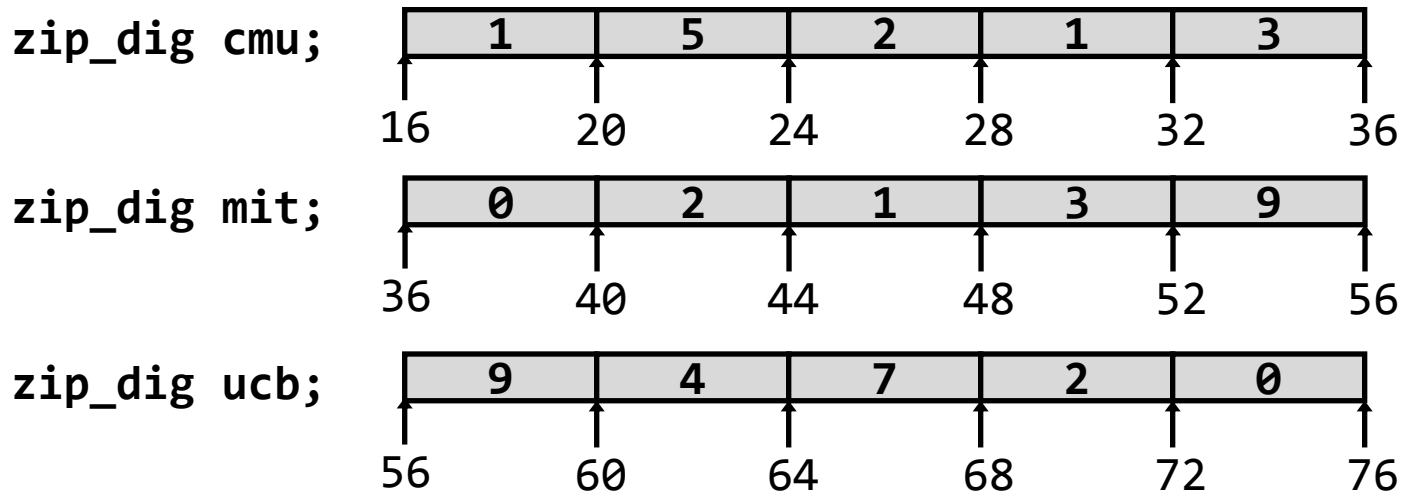


Лекция 11

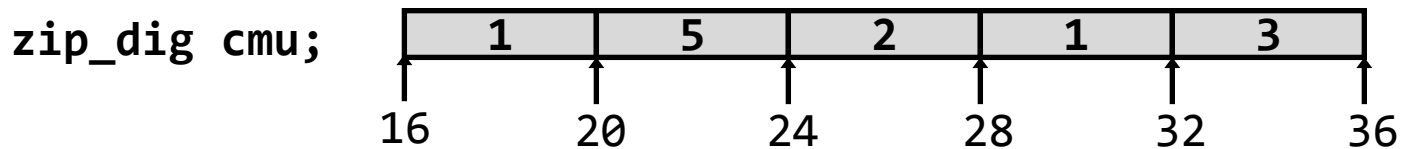
16 марта

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Объявление переменной “`zip_dig cmu`” эквивалентно “`int cmu[5]`”
- Массивы были размещены в последовательно идущих блоках памяти размером 20 байт каждый
 - В общем случае не гарантируется, что массивы будут размещены непрерывно



```
int get_digit (zip_dig z, int dig) {
    return z[dig];
}
```

```
; edx = z
; eax = dig
mov eax, dword [edx+4*eax] ; z[dig]
```

- Регистр `edx` содержит начальный (базовый) адрес массива
- Регистр `eax` содержит индекс элемента в массиве
- Адрес элемента $edx + 4 * eax$

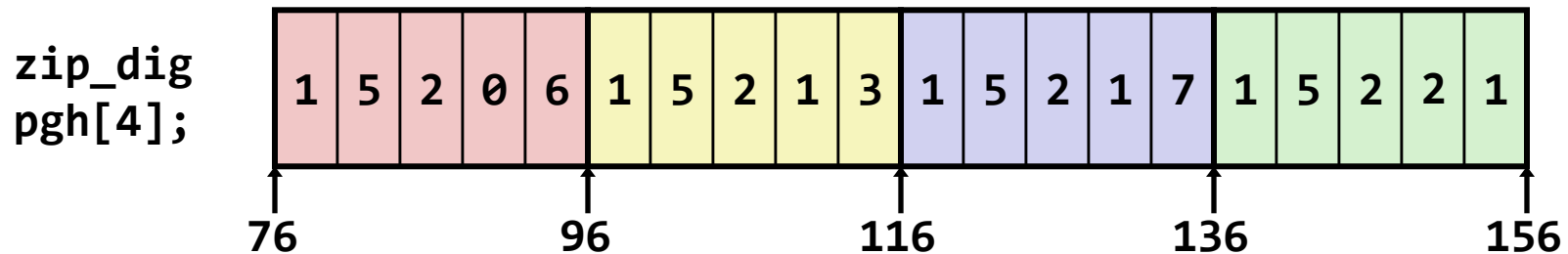
```
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```



```
void zincr_v(zip_dig z) {
    void *vz = z;
    int i = 0;
    do {
        (*((int *) (vz + i))) += 1;
        i += ISIZE;
    } while (i != ISIZE * ZLEN);
}
```

	; edx = z
mov eax, 0	; eax = i
.L4:	; loop:
add dword [edx + 4 * eax], 1	; z[i]++
add eax, 1	; i++
cmp eax, 5	; i vs. 5
jne .L4	; if (!=) goto loop

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```



- “zip_dig pgh[4]” эквивалентно “int pgh[4][5]”
 - Переменная **pgh**: массив из 4 элементов, расположенных непрерывно в памяти
 - Каждый элемент – массив из 5 **int**’ов, расположенных непрерывно в памяти
- Всегда развертывание по строкам (Row-Major)

- Объявление

$T \ A[R][C];$

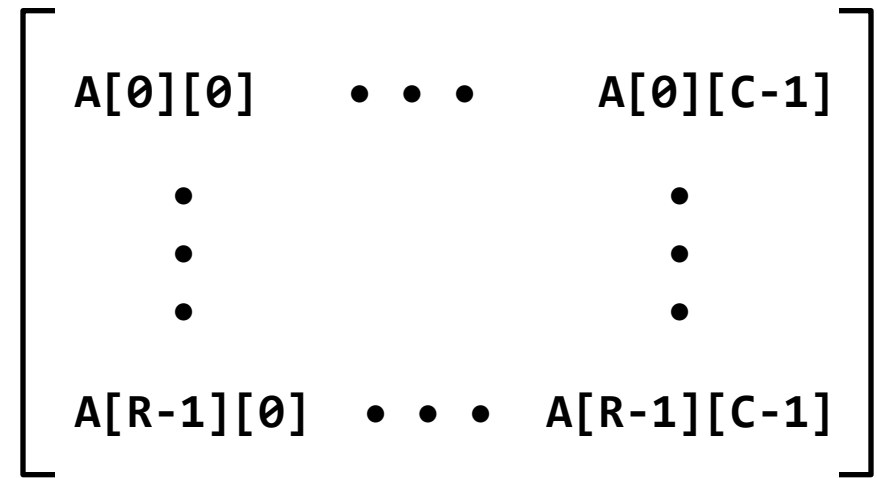
- 2D массив элементов типа T
- R строк, C столбцов
- Размер типа T – K байт

- Размер массива

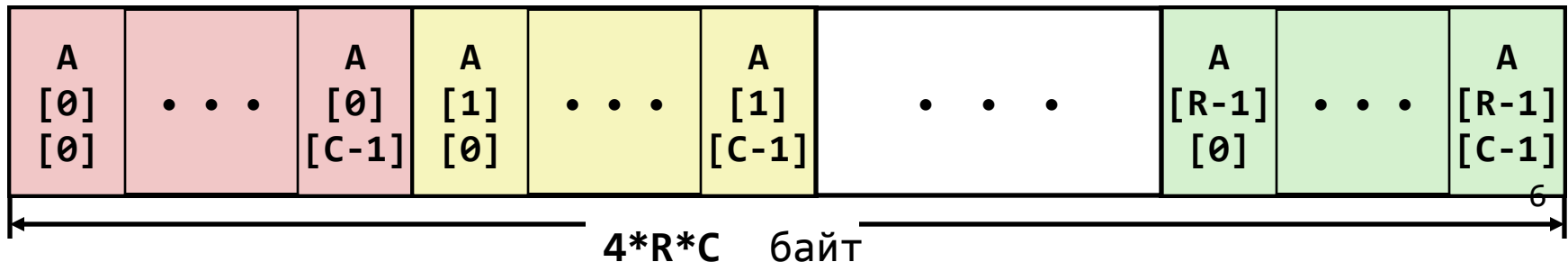
- $R * C * K$ байт

- Размещение в памяти

- Развертывание по строкам



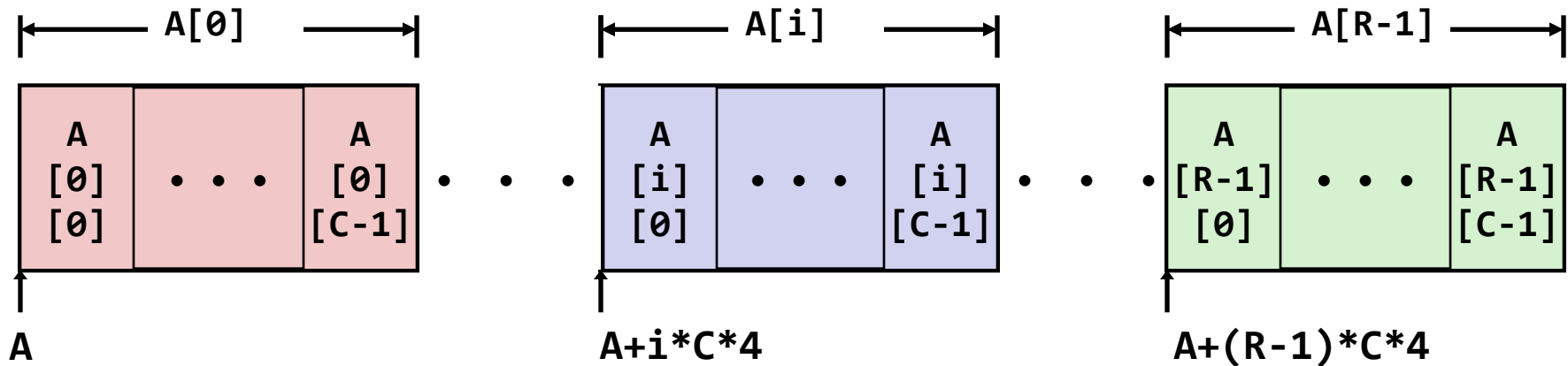
`int A[R][C];`



- Доступ к строкам

- $A[i]$ массив из C элементов
- Каждый элемент типа T требует K байт
- Начальный адрес строки с индексом i
 $A + i * (C * K)$

```
int A[R][C];
```



```
int *get_pgh_zip(int index){
    return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

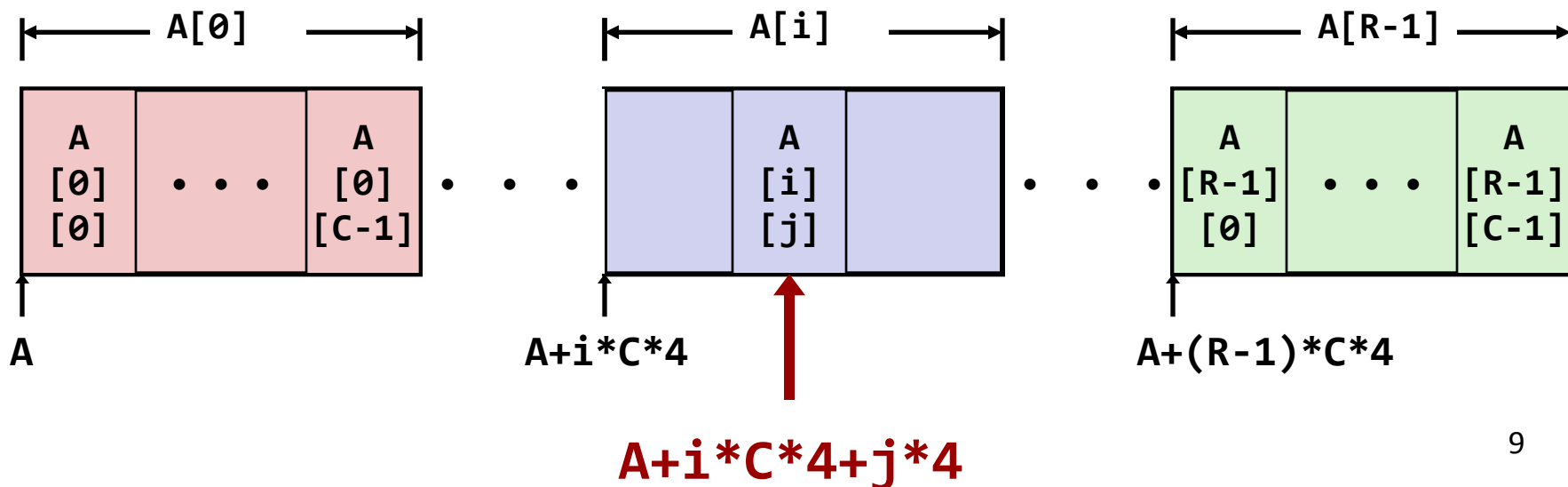
```
; eax = index
lea eax, [eax + 4 * eax] ; 5 * index
lea eax, [pgh + 4 * eax] ; pgh + (20 * index)
```

- **pgh[index]** массив из 5 int'ов
- Начальный адрес **pgh+20*index**
- Вычисляется и возвращается адрес
- Вычисление адреса в виде **pgh + 4*(index+4*index)**

- Элементы массива

- $A[i][j]$ элемент типа T , который требует K байт
- Адрес элемента $A + i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



```
int get_pgh_digit (int index, int dig) {
    return pgh[index][dig];
}
```

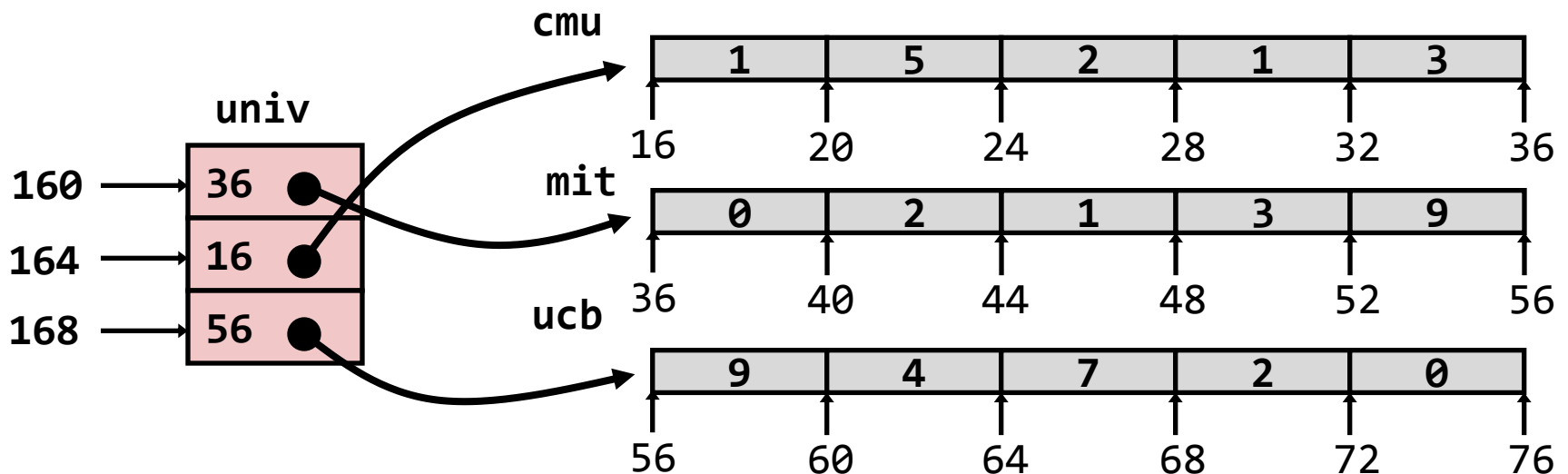
```
mov    eax, dword [ebp + 8]          ; index
lea     eax, [eax + 4 * eax]         ; 5*index
add     eax, dword [ebp + 12]        ; 5*index+dig
mov     eax, dword [pgh + 4 * eax]   ; смещение 4*(5*index+dig)
```

- $\text{pgh}[\text{index}][\text{dig}]$ – тип int
- Адрес: $\text{pgh} + 20 * \text{index} + 4 * \text{dig} =$
 $\quad \quad \quad = \text{pgh} + 4 * (5 * \text{index} + \text{dig})$
- Вычисление адреса производится как
 $\text{pgh} + 4 * ((\text{index} + 4 * \text{index}) + \text{dig})$

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Переменная `univ` представляет собой массив из 3 элементов
- Каждый элемент – указатель (размером 4 байта)
- Каждый указатель ссылается на массив из `int`'ов



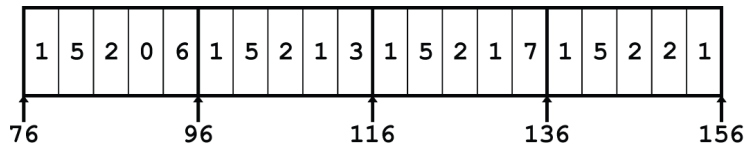
```
int get_univ_digit (int index, int dig) {  
    return univ[index][dig];  
}
```

```
mov    eax, dword [ebp + 8]          ; index  
mov    edx, dword [univ + 4 * eax]   ; p = univ[index]  
mov    eax, dword [ebp + 12]        ; dig  
mov    eax, dword [edx + 4 * eax]    ; p[dig]
```

- Доступ к элементу **Mem[Mem[univ+4*index]+4*dig]**
- Необходимо выполнить два чтения из памяти
 - Первое чтение получает указатель на одномерный массив
 - Затем второе чтение выполняет выборку требуемого элемента этого одномерного массива

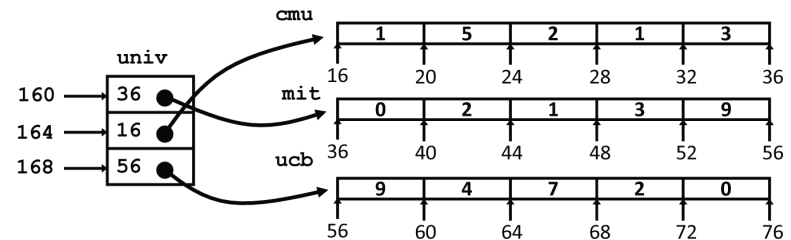
Многомерный массив

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```



Многоуровневый массив

```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```



- Значительное внешнее сходство в Си
- Существенное различие в ассемблере

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{dig}]$

$\text{Mem}[\text{Mem}[\text{univ} + 4 * \text{index}] + 4 * \text{dig}]$

Матрица N X N

- Фиксированные размерности
 - Значение N известно во время компиляции
- Динамически задаваемая размерность. Требуется явное преобразование индексов
 - Традиционный способ реализации динамических массивов
- Динамически задаваемая размерность с неявной индексацией.
 - Поддерживается последними версиями gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
    (fix_matrix a, int i, int j)
{
    return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
    (int n, int *a, int i, int j)
{
    return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele
    (int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

Матрица 16 X 16

■ Доступ к элементу матрицы

- Адрес $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* Получение элемента a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

```
mov    edx, dword [ebp + 12]    ; i
sal    edx, 6                  ; i*64
mov    eax, dword [ebp + 16]    ; j
sal    eax, 2                  ; j*4
add    eax, dword [ebp + 8]     ; a + j*4
mov    eax, dword [eax + edx]   ; *(a + j*4 + i*64)
```

Матрица n X n

■ Доступ к элементу матрицы

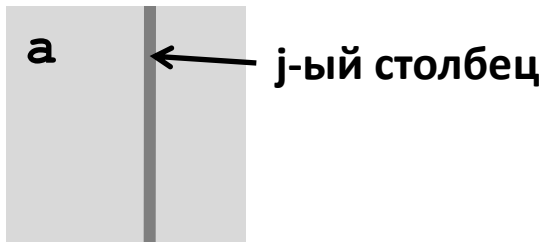
- Адрес $A + i * (C * K) + j * K$
- $C = n, K = 4$

`sizeof(a) = ?`
`sizeof(a[i]) = ?`

```
/* Получение элемента a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

```
mov    eax, dword [ebp + 8]      ; n
sal    eax, 2                   ; n*4
mov    edx, eax                 ; n*4
imul   edx, dword [ebp + 16]    ; i*n*4
mov    eax, dword [ebp + 20]    ; j
sal    eax, 2                   ; j*4
add    eax, dword [ebp + 12]    ; a + j*4
mov    eax, dword [eax + edx]   ; *(a + j*4 + i*n*4)
```


Оптимизация доступа к элементам массива



```
#define N 16  
typedef int fix_matrix[N][N];
```

- Вычисления
 - Проход по всем элементам в столбце j
- Оптимизация
 - Выборка последовательных элементов из отдельного столбца

```
/* Выборка столбца j из массива */  
void fix_column  
  (fix_matrix a, int j, int *dest)  
{  
    int i;  
    for (i = 0; i < N; i++)  
        dest[i] = a[i][j];  
}
```

Оптимизация доступа к элементам массива

• Оптимизация

– Вычисляем $\text{ajp} = \&a[i][j]$

- Начальное значение $a + 4*j$
- Шаг $4*N$

Регистр	Значение
ecx	ajp
ebx	dest
edx	i

```
/* Выборка столбца j из массива */
void fix_column
(fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

<pre>.L8: mov eax, dword [ecx] mov dword [ebx + 4 * edx], eax add edx, 1 add ecx, 64 cmp edx, 16 jne .L8</pre>	<pre>; loop: ; считываем *ajp ; сохраняем в dest[i] ; i++ ; ajp += 4*N ; i vs. N ; if !=, goto loop</pre>
--	---

Оптимизация доступа к элементам массива

– Вычисляем $ajp = \&a[i][j]$

- Начальное значение $a + 4*j$
- Шаг $4*n$

Регистр	Значение
ecx	ajp
edi	dest
edx	i
ebx	$4*n$
esi	n

```
/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

<pre>.L18: mov eax, dword [ecx] mov dword [edi + 4 * edx], eax add edx, 1 add ecx, ebx cmp esi, edx jg .L18</pre>	<pre>; loop: ; считываем *ajp ; сохраняем в dest[i] ; i++ ; ajp += 4*n ; n vs. i ; if (>) goto loop</pre>
--	--

Оптимизация доступа к элементам массива

– Изменение направления прохода по циклу

- Выход из цикла по нулевому счетчику
- Шаг отрицательный
- Меняются начальные значения указателей
- Достаточно вывести к нулю один из индексов

```
/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest) {

    int i;
    for (i = n-1; i >=0; i--) {
        dest[i] = a[i][j];
    }
}
```

.L18:

```
mov     eax, dword [ecx]
mov     dword [edi + 4 * edx], eax
add     edx, 1
add     ecx, ebx
cmp     esi, edx
jg      .L18
```

; loop:

```
; считываем *ajp
; сохраняем в dest[i]
; i++
; ajp += 4*n
; n vs. i
; if (>) goto loop
```

Оптимизация доступа к элементам массива

Регистр	Начальное значение
ecx	$a + 4 * n * (n - 1) + 4 * j$
edi	<code>dest - 4</code>
edx	<code>n</code>
ebx	$4 * n$
esi	освободился

```

/* Выборка столбца j из массива */
void var_column
(int n, int a[n][n],
 int j, int *dest) {

    int i;
    dest--;
    for (i = n; i != 0; i--)
        dest[i] = a[i-1][j];
}

```

Машинно-зависимая оптимизация

<pre> .L18: mov eax, dword [ecx] mov dword [edi + 4 * edx], eax sub ecx, ebx sub edx, 1 jnz .L18 </pre>	<pre> ; loop: ; считываем *(ajp+...) ; сохраняем в dest[i] ; ajp -= 4*n ; i-- ; if (!=) goto loop </pre>
---	--

Обратная задача

M = ?, N = ?

```

; пролог функции пропущен
mov  ecx, dword [ebp + 8]      ; 1
mov  edx, dword [ebp + 12]    ; 2
lea  eax, [8 * ecx]           ; 3
sub  eax, ecx                  ; 4
add  eax, edx                  ; 5
lea  edx, [edx + 4 * edx]     ; 6
add  edx, ecx                  ; 7
mov  eax, dword [m1 + 4 * eax] ; 8
add  eax, dword [m2 + 4 * edx] ; 9
; эпилог функции пропущен

```

```

int m1[M][N];
int m2[N][M];

int sum_element(int i, int j) {
    return m1[i][j] + m2[j][i];
}

```