Introduction to Computer Systems

15-213, fall 2009

7th Lecture, Sep. 14th

Instructors:

Majd Sakr and Khaled Harras

Last Time

- **For loops**
  - for loop → while loop → do-while loop → goto version
  - for loop → while loop → goto "jump to middle" version

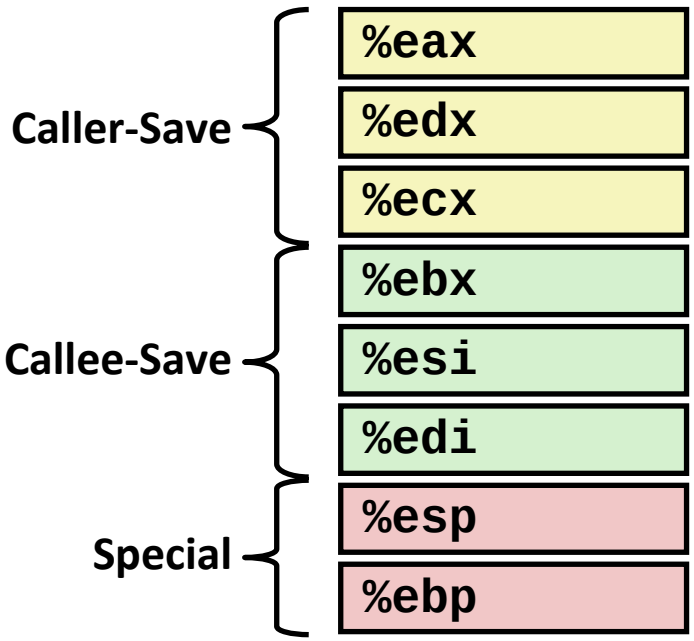
- **Switch statements**
  - Jump tables: `jmp *.L62(,%edx,4)`
  - Decision trees (not shown)

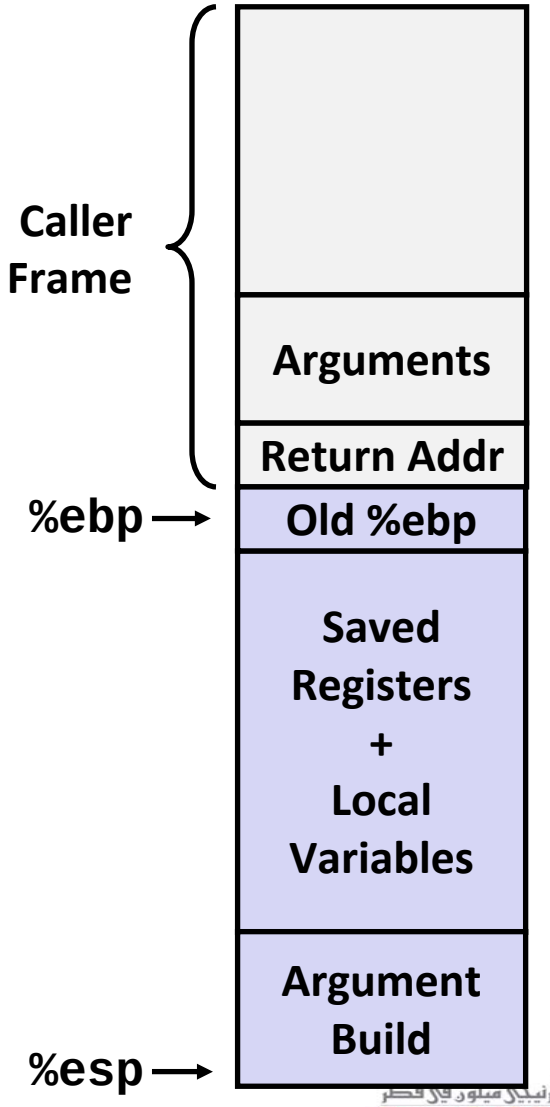
Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

Last Time

- **Procedures (IA32)**
  - call / return
  - %esp, %ebp
  - local variables
  - recursive functions





Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**

Basic Data Types

Integral

- Stored & operated on in general (integer) registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	**b**	1	**[unsigned] char**
word	**w**	2	**[unsigned] short**
double word	**l**	4	**[unsigned] int**
quad word	**q**	8	**[unsigned] long int** (x86-64)

Floating Point

- Stored & operated on in floating point registers

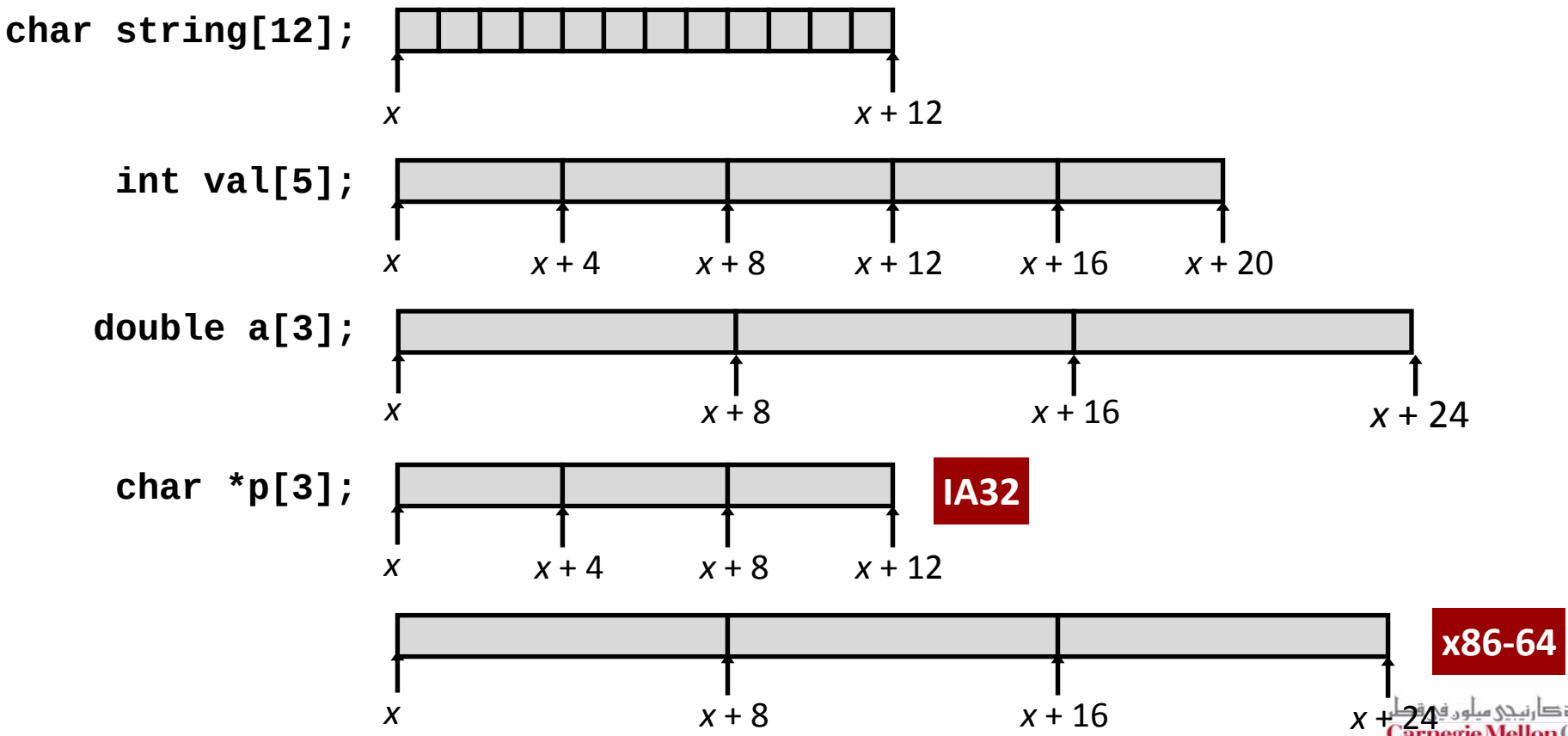
Intel	GAS	Bytes	C
Single	**s**	4	**float**
Double	**l**	8	**double**
Extended	**t**	10/12/16	**long double**

Array Allocation

- **Basic Principle**

  T **A[***L***];**

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof(***T***)** bytes

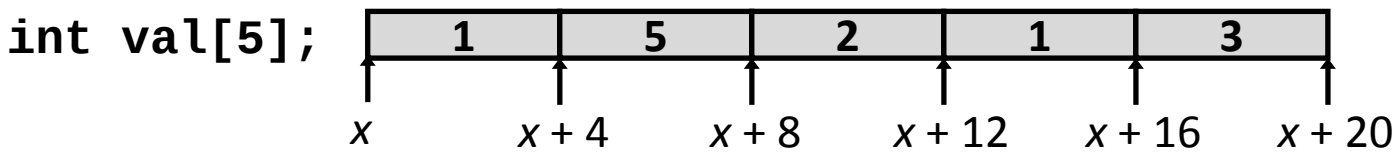


Array Access

- **Basic Principle**

  T **A[***L***];**

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**



- **Reference** **Type** **Value**

  val[4]
  val
  val+1
  &val[2]
  val[5]
  ***(val+1)**
  val + *i*

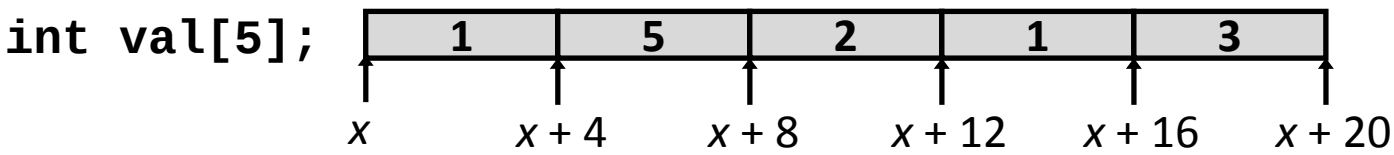
Will disappear
Blackboard?

Array Access

- **Basic Principle**

  T **A[***L***];**

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

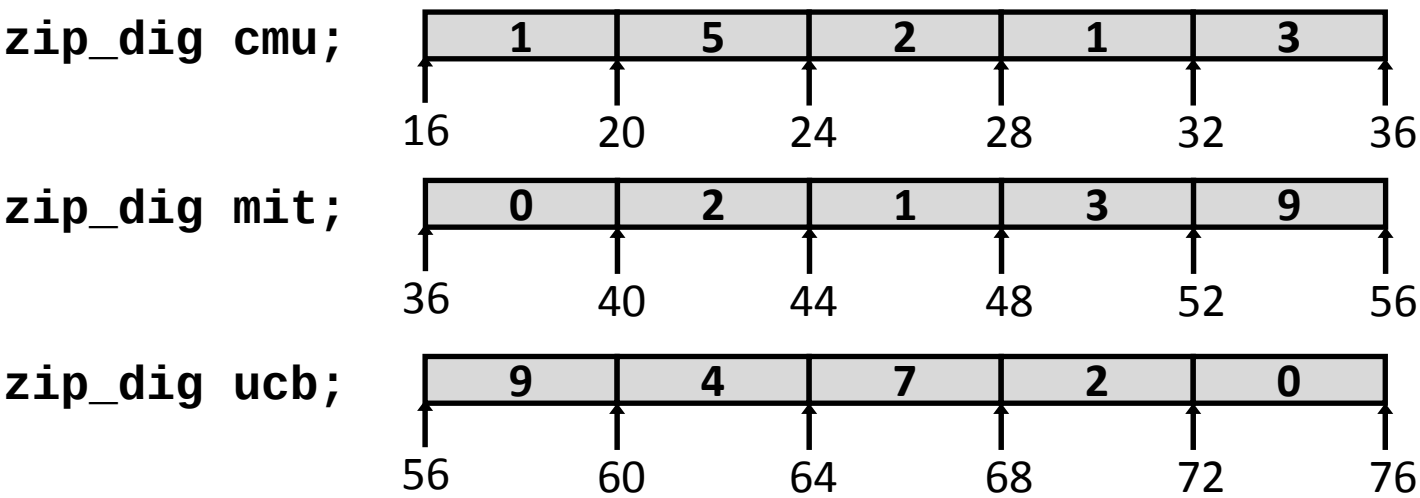


- **Reference** **Type** **Value**

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Array Example

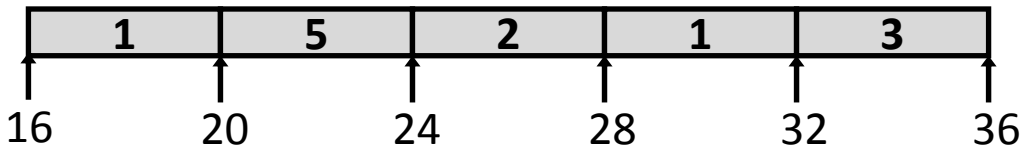
```
typedef int zip_dig[5];

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



- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig cmu;
```




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

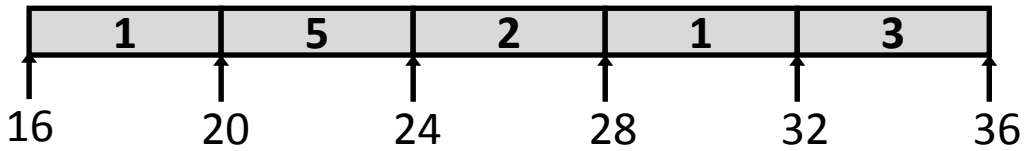
IA32

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax  # z[dig]
```

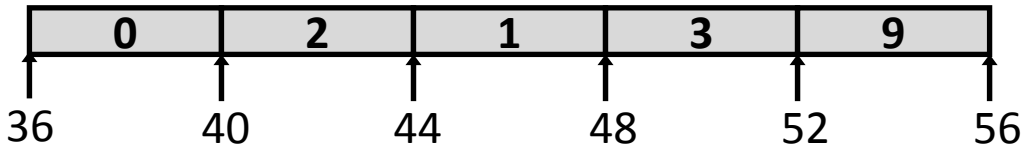
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx,%eax,4)

Referencing Examples

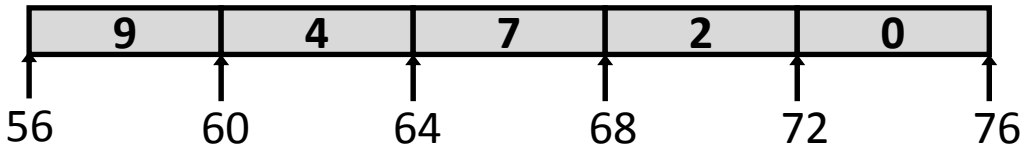
```
zip_dig cmu;
```




```
zip_dig mit;
```




```
zip_dig ucb;
```



- **Reference** **Address** **Value** **Guaranteed?**

  `mit[3]`

  `mit[5]`

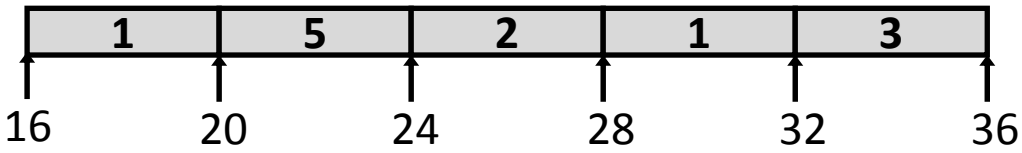
  `mit[-1]`

  `cmu[15]`

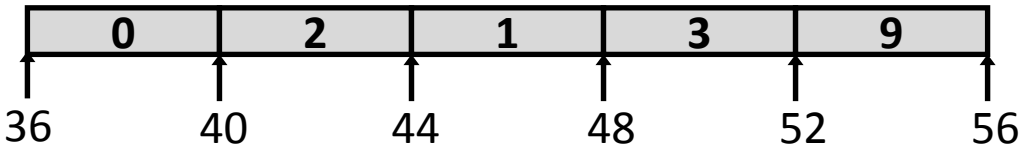
Will disappear
Blackboard?

Referencing Examples

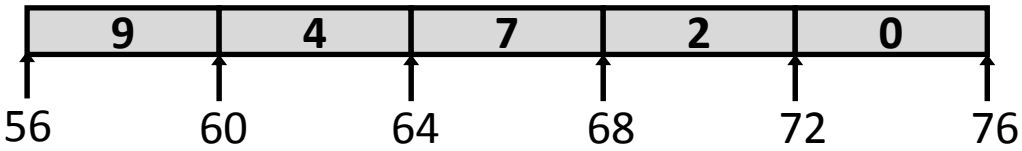
```
zip_dig cmu;
```




```
zip_dig mit;
```




```
zip_dig ucb;
```



Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	Yes
`mit[5]`	`36 + 4* 5 = 56`	`9`	No
`mit[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

- No bound checking
- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Loop Example

- **Original**

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- **Transformed**
  - As generated by GCC
  - Eliminate loop variable `i`
  - Convert array code to pointer code
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

Array Loop Implementation (IA32)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax
  leal 16(%ecx),%ebx
.L59:
  leal (%eax,%eax,4),%edx
  movl (%ecx),%eax
  addl $4,%ecx
  leal (%eax,%edx,2),%eax
  cmpl %ebx,%ecx
  jle .L59
```

Will disappear Blackboard?

Array Loop Implementation (IA32)

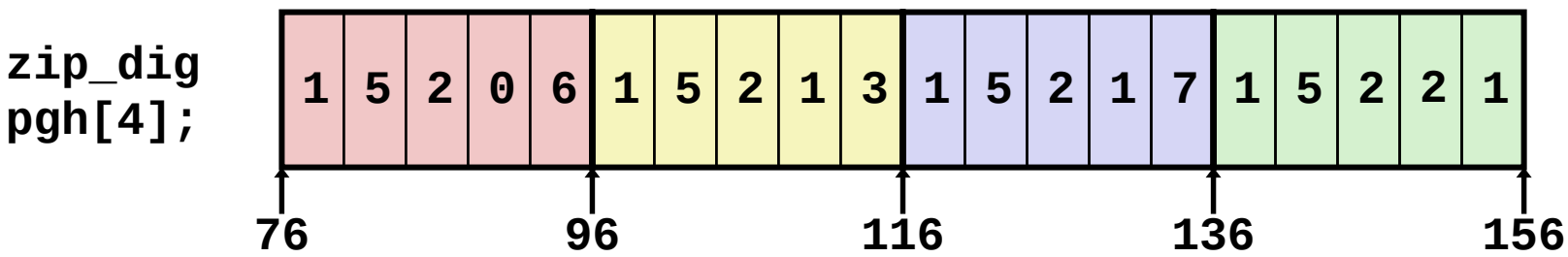
- **Registers**
  - `%ecx z`
  - `%eax zi`
  - `%ebx zend`
- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
  - `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
   # %ecx = z
   xorl %eax,%eax            # zi = 0
   leal 16(%ecx),%ebx        # zend  = z+4
.L59:
   leal (%eax,%eax,4),%edx   # 5*zi
   movl (%ecx),%eax          # *z
   addl $4,%ecx              # z++
   leal (%eax,%edx,2),%eax   # zi = *z + 2*(5*zi)
   cmpl %ebx,%ecx            # z : zend
   jle .L59                  # if <= goto loop
```

Nested Array Example

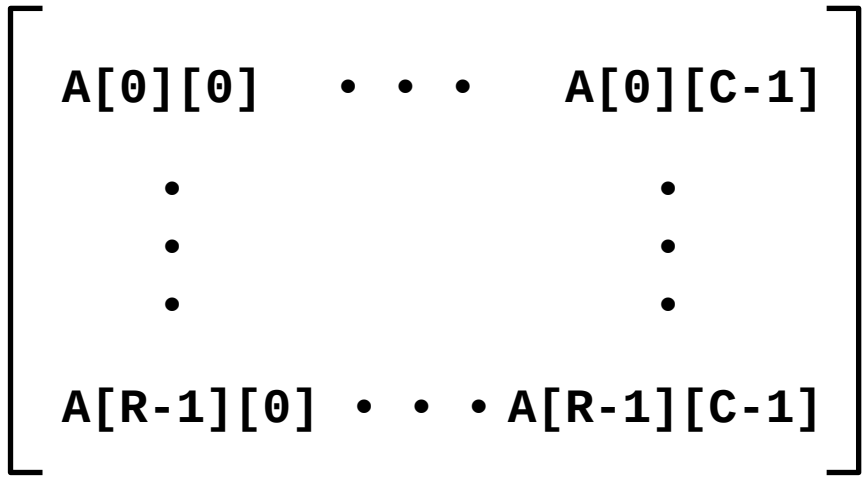
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



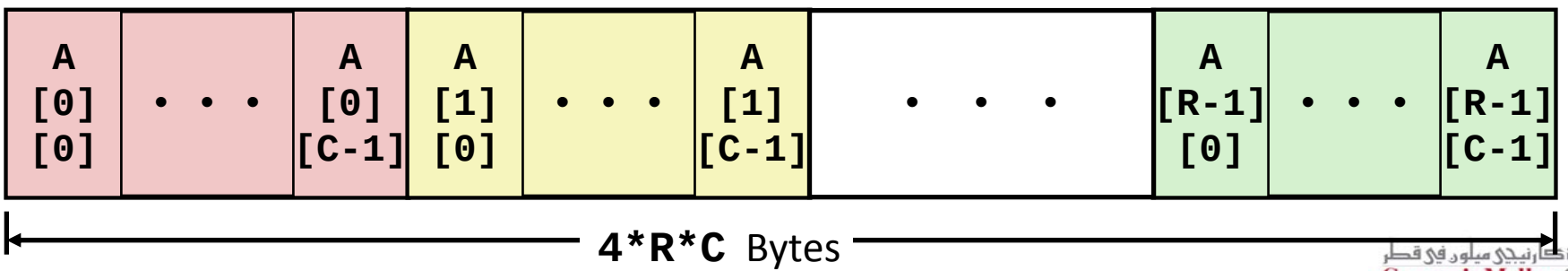
- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **pgh**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

Multidimensional (Nested) Arrays

- **Declaration**
  - *T* **A**[*R*][*C*];
    - 2D array of data type *T*
    - *R* rows, *C* columns
    - Type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering



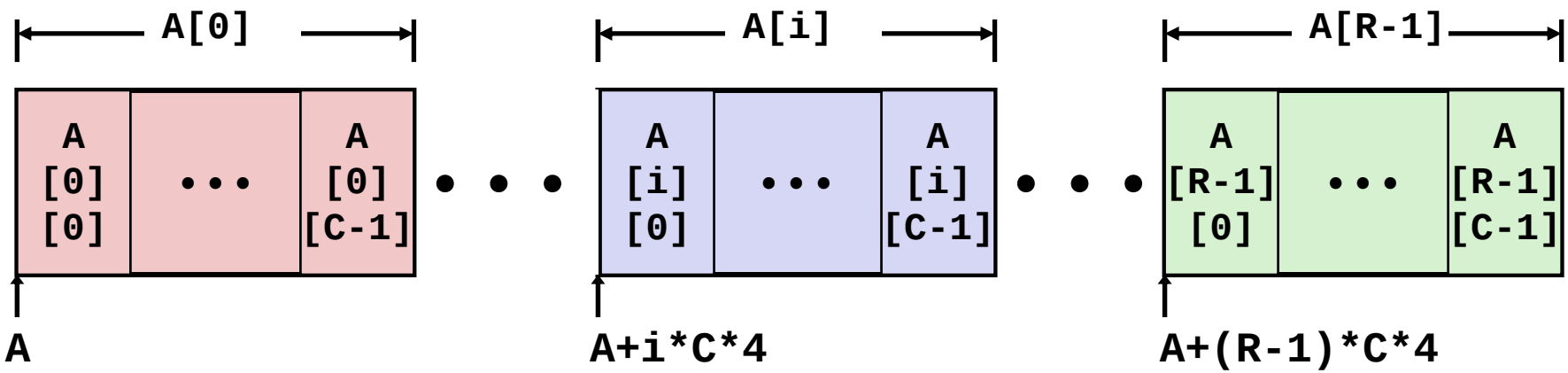
`int A[R][C];`



Nested Array Row Access

- **Row Vectors**
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A + `** *i* * (*C* * *K*)

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



Nested Array Row Access Code

```
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 }};
```

- What data type is `pgh[index]`?
- What is its starting address?

```
  # %eax = index
   leal (%eax,%eax,4),%eax
   leal pgh(,%eax,4),%eax
```

Will disappear Blackboard?

Nested Array Row Access Code

```
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 }};
```

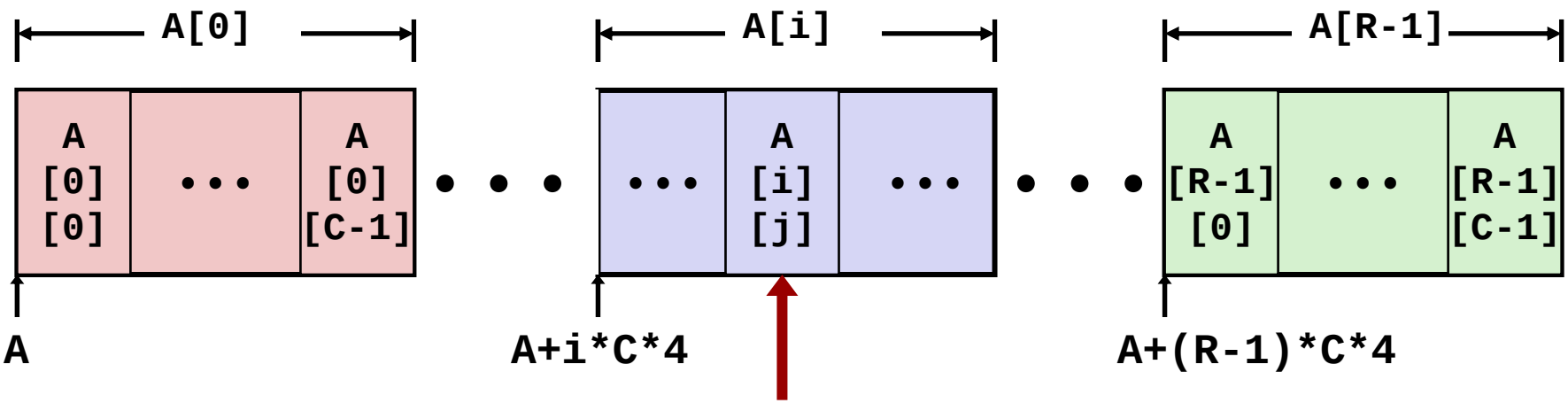
```
  # %eax = index
   leal (%eax,%eax,4),%eax # 5 * index
   leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

- **Row Vector**
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- **IA32 Code**
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

Nested Array Row Access

- **Array Elements**
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address **A** + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

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



Nested Array Element Access Code

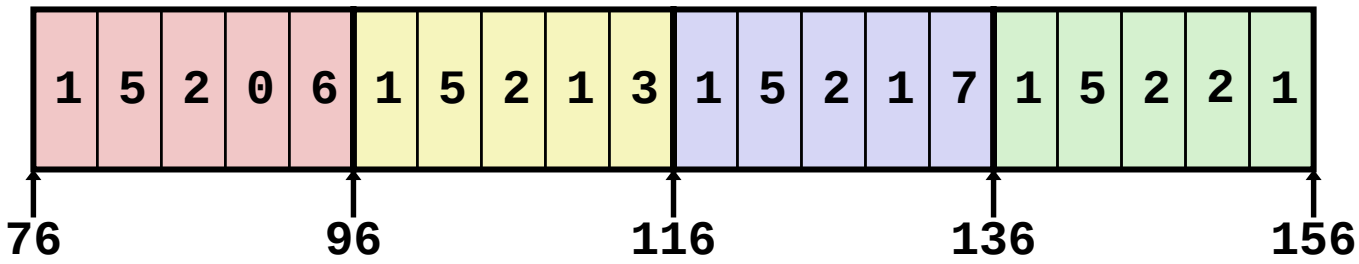
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
 # %ecx = dig
 # %eax = index
 leal 0(,%ecx,4),%edx          # 4*dig
 leal (%eax,%eax,4),%eax       # 5*index
 movl pgh(%edx,%eax,4),%eax    # *(pgh + 4*dig + 20*index)
```

- **Array Elements**
  - `pgh[index][dig]` is `int`
  - Address: `pgh + 20*index + 4*dig`
- **IA32 Code**
  - Computes address `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference

Strange Referencing Examples

```
zip_dig
pgh[4];
```

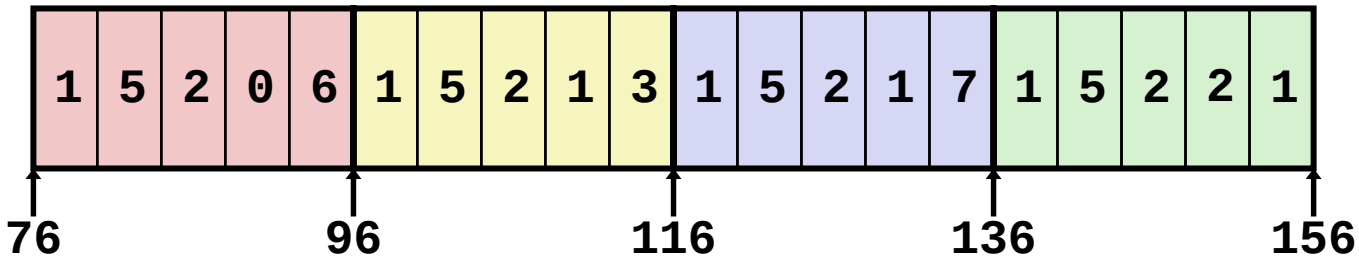


- Reference Address Value Guaranteed?
  - `pgh[3][3]`
  - `pgh[2][5]`
  - `pgh[2][-1]`
  - `pgh[4][-1]`
  - `pgh[0][19]`
  - `pgh[0][-1]`

Will disappear

Strange Referencing Examples

```
zip_dig
pgh[4];
```



Reference	Address	Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3 = 148`	2	Yes
`pgh[2][5]`	`76+20*2+4*5 = 136`	1	Yes
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	3	Yes
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	1	Yes
`pgh[0][19]`	`76+20*0+4*19 = 152`	1	Yes
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	??	No

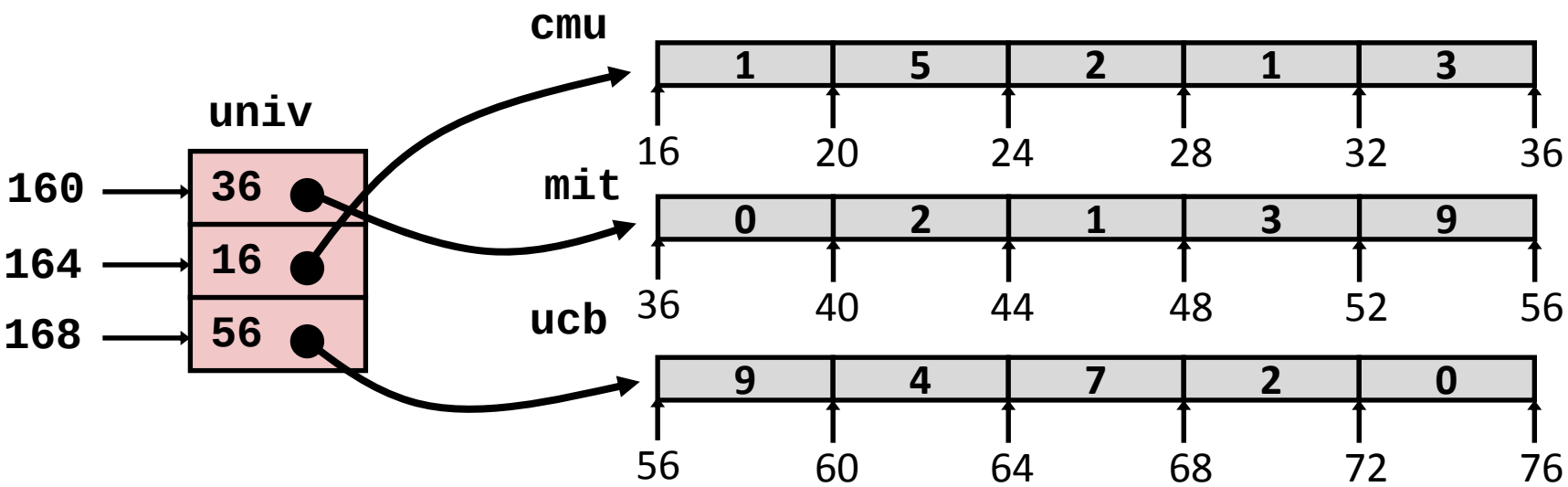
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

```
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};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s



Element Access in Multi-Level Array

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

```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx
  movl univ(%edx),%edx
  movl (%edx,%eax,4),%eax
```

Will disappear Blackboard?

Element Access in Multi-Level Array

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

```
   # %ecx = index
   # %eax = dig
   leal 0(,%ecx,4),%edx     # 4*index
   movl univ(%edx),%edx     # Mem[univ+4*index]
   movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

- **Computation (IA32)**
  - Element access **`Mem[Mem[univ+4*index]+4*dig]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

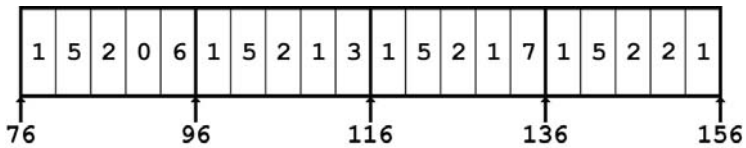
Array Element Accesses

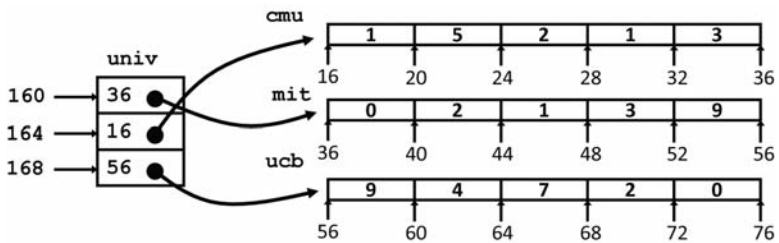
Nested array

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

Multi-level array

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



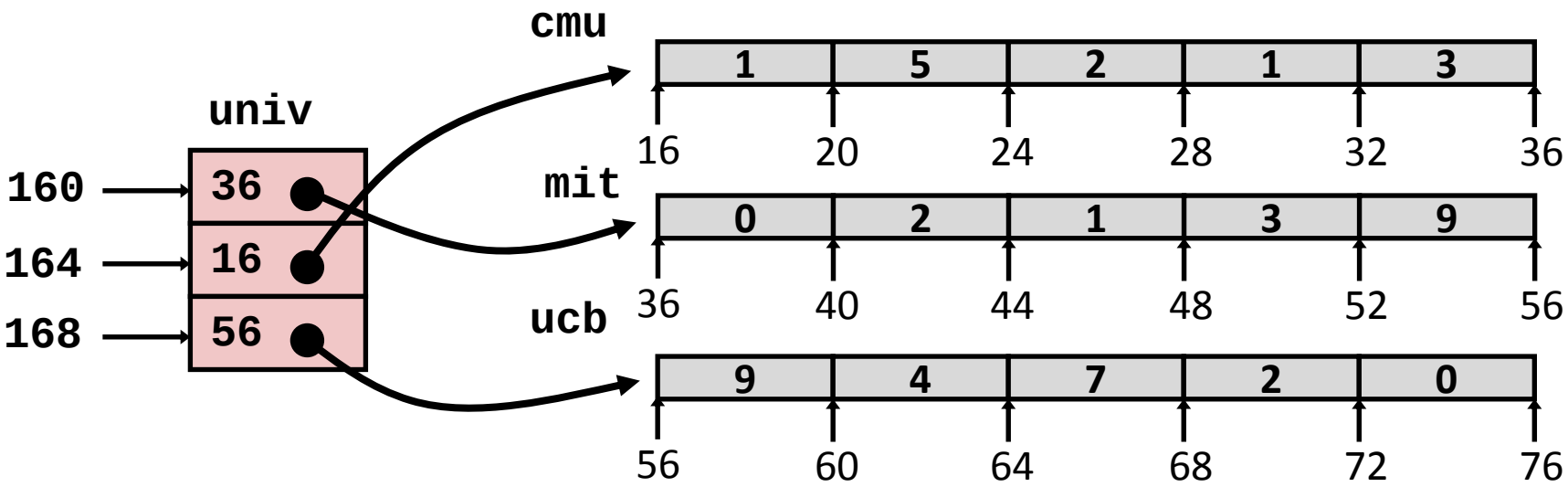


Access looks similar, but element:

`Mem[pgh+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

Strange Referencing Examples

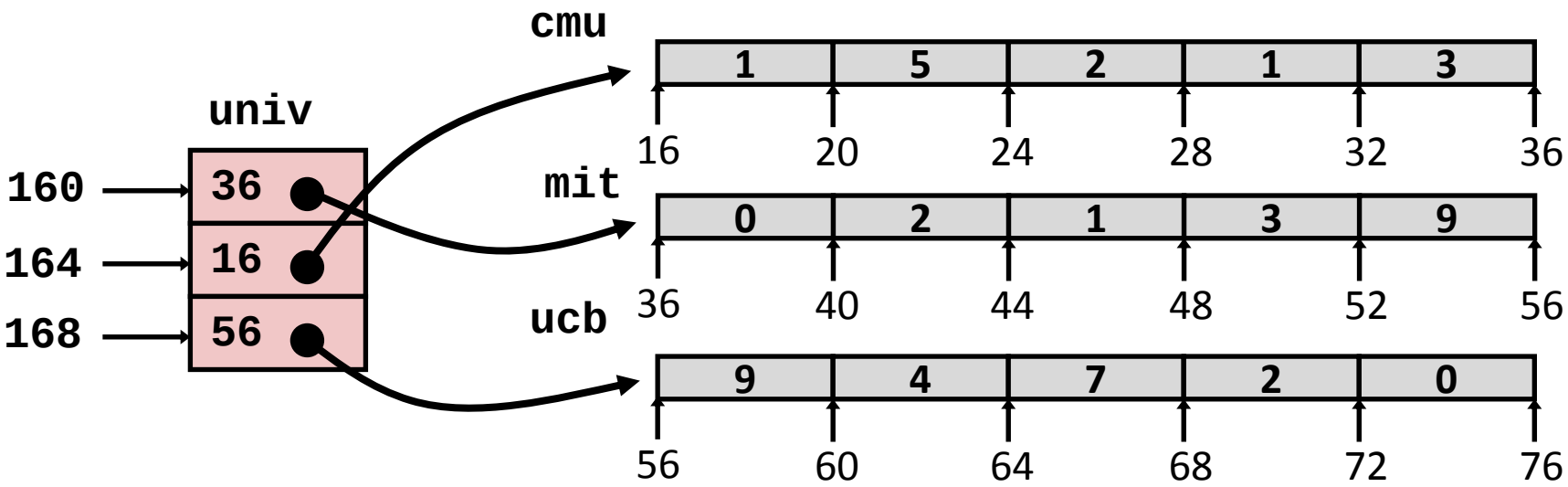


- Reference Address Value Guaranteed?

  `univ[2][3]`
  `univ[1][5]`
  `univ[2][-1]`
  `univ[3][-1]`
  `univ[1][12]`

  Will disappear

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	`2`	Yes
`univ[1][5]`	`16+4*5 = 36`	`0`	No
`univ[2][-1]`	`56+4*-1 = 52`	`9`	No
`univ[3][-1]`	`??`	`??`	No
`univ[1][12]`	`16+4*12 = 64`	`7`	No

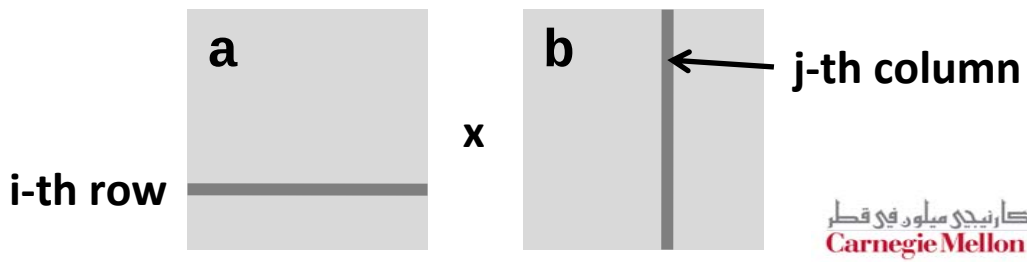
- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

Using Nested Arrays

- **Strengths**
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
  - Avoids multiply in index computation
- **Limitation**
  - Only works for fixed array size

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

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```



Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size
- **Programming**
  - Must do index computation explicitly
- **Performance**
  - Accessing single element costly
  - Must do multiplication

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

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

```
movl 12(%ebp),%eax       # i
movl 8(%ebp),%edx        # a
imull 20(%ebp),%eax      # n*i
addl 16(%ebp),%eax       # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```

Dynamic Array Multiplication

- **Without Optimizations**
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

```
/* Compute element i,k of
   variable matrix product */
int var_prod_ele
  (int *a, int *b,
   int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to **-O2**
- **Code Motion**
  - Expression **i*n** can be computed outside loop
- **Strength Reduction**
  - Incrementing **j** has effect of incrementing **j*n+k** by **n**
- **Operations count**
  - 4 adds, 1 mult
- **Compiler can optimize regular access patterns**

```
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types
- **Accessing Structure Member**

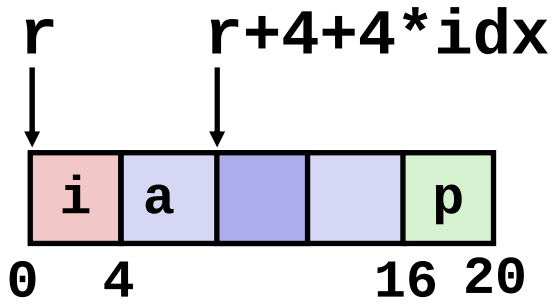
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Structure Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

```
int *find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

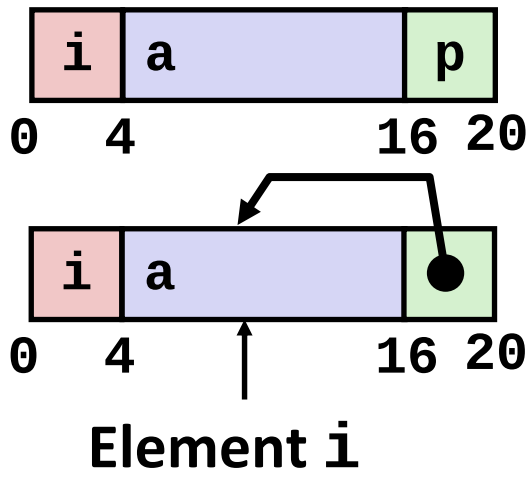
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Structure Referencing (Cont.)

- C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```




```
 # %edx = r
 movl (%edx),%ecx         # r->i
 leal 0(,%ecx,4),%eax     # 4*(r->i)
 leal 4(%edx,%eax),%eax   # r+4+4*(r->i)
 movl %eax,16(%edx)       # Update r->p
```