Organisation und Architektur von Rechnern

Lecture 07

Instructor:

Reinhard v. Hanxleden

http://www.informatik.uni-kiel.de/rtsys/teaching/v-sysinf2

These slides are used with kind permission from the Carnegie Mellon University

The 5 Minute Review Session

1. **How do we implement while-loops in assembler?**

2. **How do we implement for-loops in assembler?**

3. **How do we implement switch statements in assembler?**

4. **What is the stack?**

5. **What assembler operations affect the stack?**

Another 5 Minute Review Session

1. Why do we use a stack to implement function calls in C?

2. What is a stack frame?

3. What does a stack frame consist of?

4. What delineates the stack frame?

5. How does the callee access the calling arguments?

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)
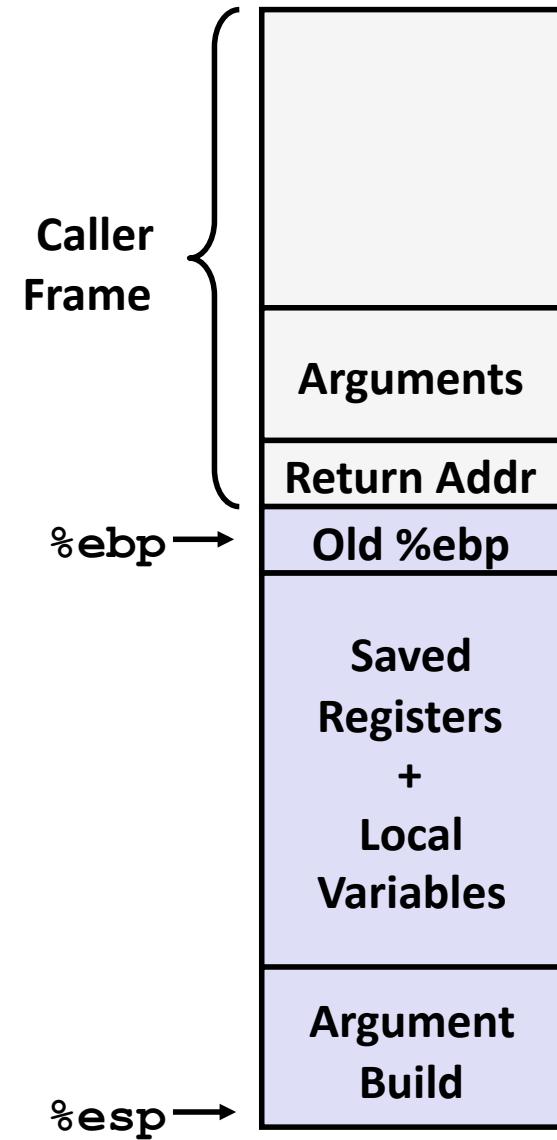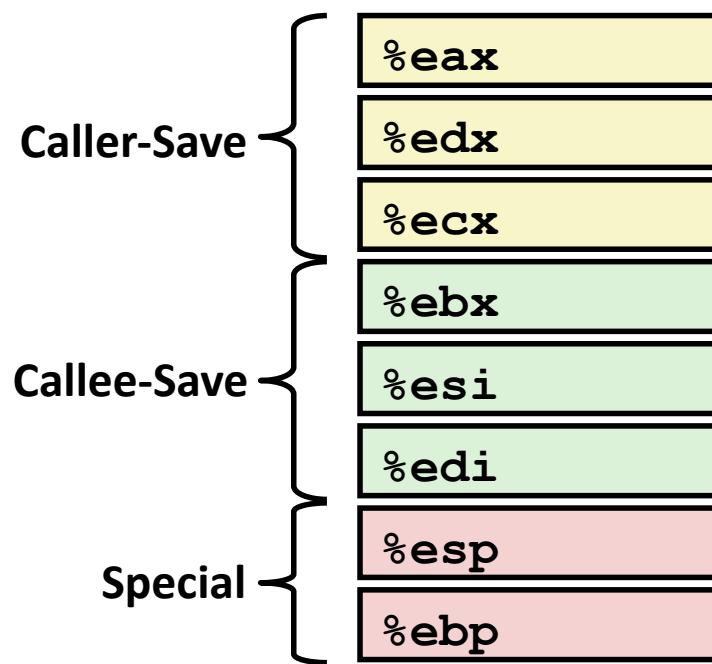
Jump table

```
.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

Caller-Save:
- %eax
- %edx
- %ecx

Callee-Save:
- %ebx
- %esi
- %edi

Special:
- %esp
- %ebp

Caller Frame:
- Arguments
- Return Addr

%ebp →
- Old %ebp
- Saved Registers + Local Variables
- Argument Build

%esp →

Today

- **Procedures (x86-64)**
- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**

x86-64 Integer Registers

%rax	%eax	**%r8**	%r8d
%rbx	%ebx	**%r9**	%r9d
%rcx	%ecx	**%r10**	%r10d
%rdx	%edx	**%r11**	%r11d
%rsi	%esi	**%r12**	%r12d
%rdi	%edi	**%r13**	%r13d
%rsp	%esp	**%r14**	%r14d
%rbp	%ebp	**%r15**	%r15d

- Twice the number of registers

- Accessible as 8, 16, 32, 64 bits

x86-64 Integer Registers

`%rax`	Return value		`%r8`	Argument #5
`%rbx`	Callee saved		`%r9`	Argument #6
`%rcx`	Argument #4		`%r10`	Callee saved
`%rdx`	Argument #3		`%r11`	Used for linking
`%rsi`	Argument #2		`%r12`	C: Callee saved
`%rdi`	Argument #1		`%r13`	Callee saved
`%rsp`	Stack pointer		`%r14`	Callee saved
`%rbp`	Callee saved		`%r15`	Callee saved

x86-64 Registers

- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well

- **All references to stack frame via stack pointer**
  - Eliminates need to update `%ebp/%rbp`

- **Other Registers**
  - 6+1 callee saved
  - 2 or 3 have special uses

x86-64 Long Swap

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```
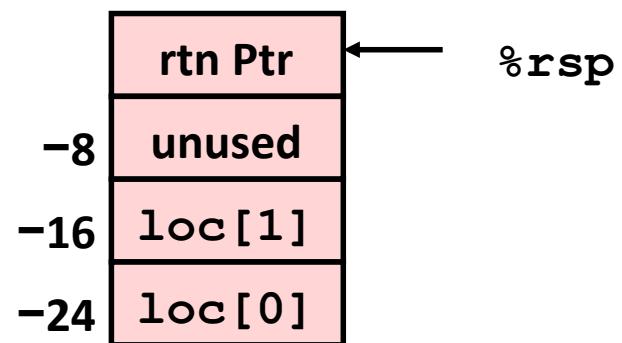
- **Operands passed in registers**
  - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  - 64-bit pointers
- **No stack operations required (except `ret`)**
- **Avoiding stack**
 - Can hold all local information in registers

x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq  (%rdi), %rax
  movq  %rax, -24(%rsp)
  movq  (%rsi), %rax
  movq  %rax, -16(%rsp)
  movq  -16(%rsp), %rax
  movq  %rax, (%rdi)
  movq  -24(%rsp), %rax
  movq  %rax, (%rsi)
  ret
```

- **Avoiding Stack Pointer Change**
  - Can hold all information within small window beyond stack pointer

rtn Ptr	← %rsp
−8	**unused**
−16	`loc[1]`
−24	`loc[0]`

x86-64 NonLeaf without Stack Frame

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele_se
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    scount++;
}
```

- No values held while swap being invoked

- No callee save registers needed

```
swap_ele_se:
  movslq %esi,%rsi            # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi  # &a[i]
  leaq   8(%rdi), %rsi        # &a[i+1]
  call   swap                 # swap()
  incq   scount(%rip)         # scount++;
  ret
```

x86-64 Call using Jump

```c
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

```
swap_ele:
  movslq %esi,%rsi
  leaq   (%rdi,%rsi,8), %rdi
  leaq   8(%rdi), %rsi
  jmp    swap
```

Will disappear Blackboard?

x86-64 Call using Jump

```
long scount = 0;

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- When swap executes `ret`, it will return from `swap_ele`
- Possible since swap is a "tail call"
  (no instructions afterwards)

```
swap_ele:
  movslq %esi,%rsi            # Sign extend i
  leaq    (%rdi,%rsi,8), %rdi # &a[i]
  leaq    8(%rdi), %rsi       # &a[i+1]
  jmp     swap                # swap()
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}
```

- **Keeps values of `a` and `i` in callee save registers**

- **Must set up stack frame to save these registers**

```
swap_ele_su:
  movq     %rbx, -16(%rsp)
  movslq   %esi,%rbx
  movq     %r12, -8(%rsp)
  movq     %rdi, %r12
  leaq     (%rdi,%rbx,8), %rdi
  subq     $16, %rsp
  leaq     8(%rdi), %rsi
  call     swap
  movq     (%r12,%rbx,8), %rax
  addq     %rax, sum(%rip)
  movq     (%rsp), %rbx
  movq     8(%rsp), %r12
  addq     $16, %rsp
  ret
```

Blackboard?

Understanding x86-64 Stack Frame

```
swap_ele_su:
  movq     %rbx, -16(%rsp)        # Save %rbx
  movslq   %esi,%rbx              # Extend & save i
  movq     %r12, -8(%rsp)         # Save %r12
  movq     %rdi, %r12             # Save a
  leaq     (%rdi,%rbx,8), %rdi    # &a[i]
  subq     $16, %rsp              # Allocate stack frame
  leaq     8(%rdi), %rsi          #     &a[i+1]
  call     swap                   # swap()
  movq     (%r12,%rbx,8), %rax    # a[i]
  addq     %rax, sum(%rip)        # sum += a[i]
  movq     (%rsp), %rbx           # Restore %rbx
  movq     8(%rsp), %r12          # Restore %r12
  addq     $16, %rsp              # Deallocate stack frame
  ret
```

Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)        # Save %rbx

    movq    %r12, -8(%rsp)         # Save %r12


    subq    $16, %rsp              # Allocate stack frame




    movq    (%rsp), %rbx           # Restore %rbx
    movq    8(%rsp), %r12          # Restore %r12
    addq    $16, %rsp              # Deallocate stack frame
```


%rsp →	rtn addr
-8	%r12
-16	%rbx

	rtn addr
+8	%r12
%rsp →	%rbx
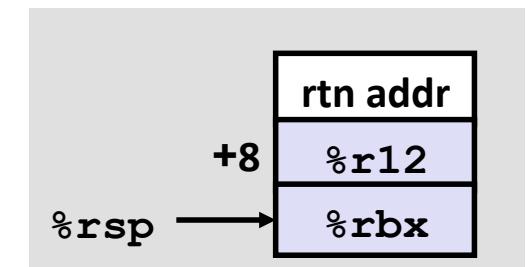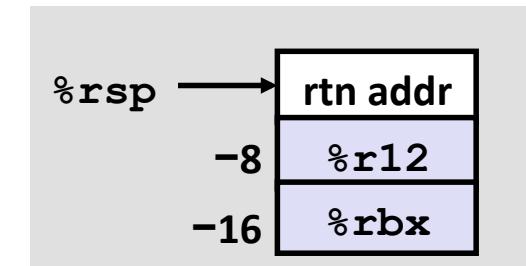
Interesting Features of Stack Frame

- **Allocate entire frame at once**

  - All stack accesses can be relative to `%rsp`

  - Do by decrementing stack pointer

  - Can delay allocation, since safe to temporarily use red zone

- **Simple deallocation**

  - Increment stack pointer

  - No base/frame pointer needed

x86-64 Procedure Summary

- **Heavy use of registers**
  - Parameter passing
  - More temporaries since more registers

- **Minimal use of stack**
  - Sometimes none
  - Allocate/deallocate entire block

- **Many tricky optimizations**
  - What kind of stack frame to use
  - Calling with jump
  - Various allocation techniques

Today

- **Procedures (x86-64)**

- **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**
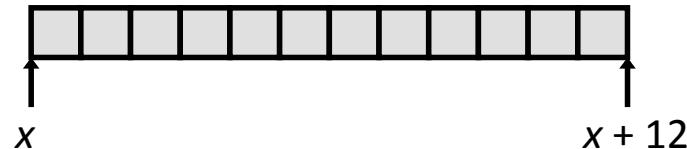
Array Allocation

- **Basic Principle**

  T `A[`*L*`];`
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * `sizeof`(*T*) bytes
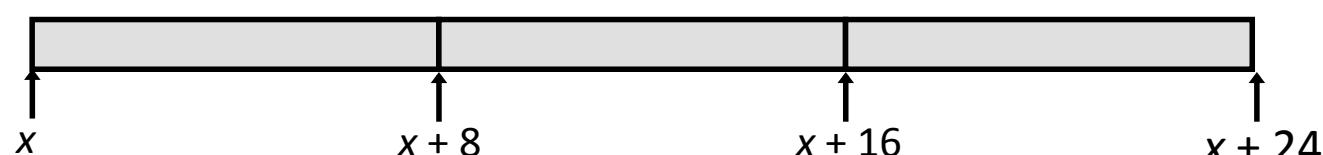
`char string[12];`
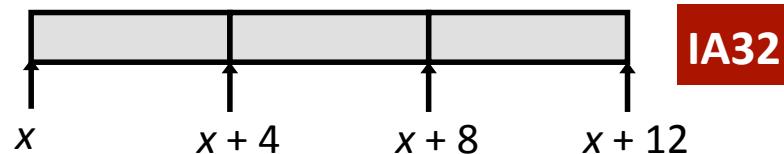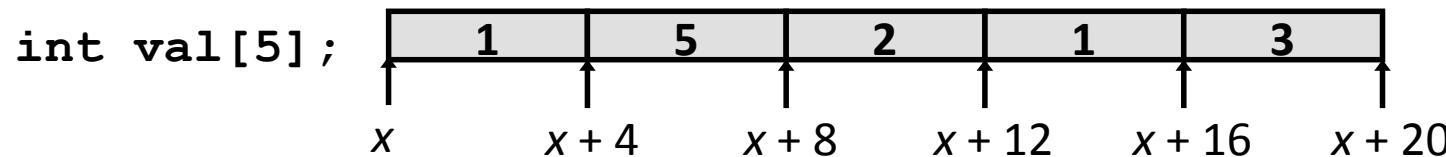
x $\qquad$ $x+12$

`int val[5];`

x $\qquad$ $x+4$ $\qquad$ $x+8$ $\qquad$ $x+12$ $\qquad$ $x+16$ $\qquad$ $x+20$

`double a[3];`

x $\qquad$ $x+8$ $\qquad$ $x+16$ $\qquad$ $x+24$

`char *p[3];`

IA32

x $\qquad$ $x+4$ $\qquad$ $x+8$ $\qquad$ $x+12$

x86-64

x $\qquad$ $x+8$ $\qquad$ $x+16$ $\qquad$ $x+24$

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**

  `int val[5];`

1	5	2	1	3

  x $x + 4$ $x + 8$ $x + 12$ $x + 16$ $x + 20$

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

  `val[4]`

  `val`

  `val+1`

  `&val[2]`

  `val[5]`

  `*(val+1)`

  `val + i`
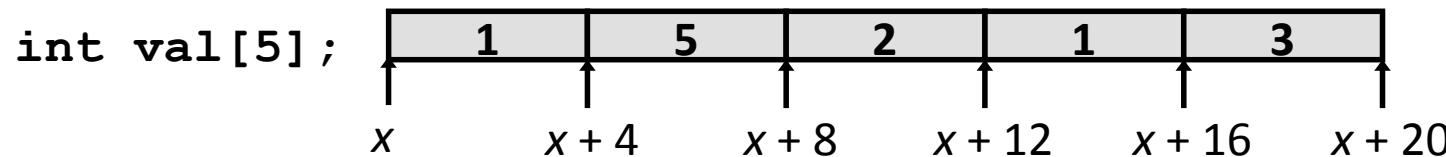
Will disappear
Blackboard?

23

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 $T*$

`int val[5];`

1	5	2	1	3
x	$x + 4$	$x + 8$	$x + 12$	$x + 16$	$x + 20$

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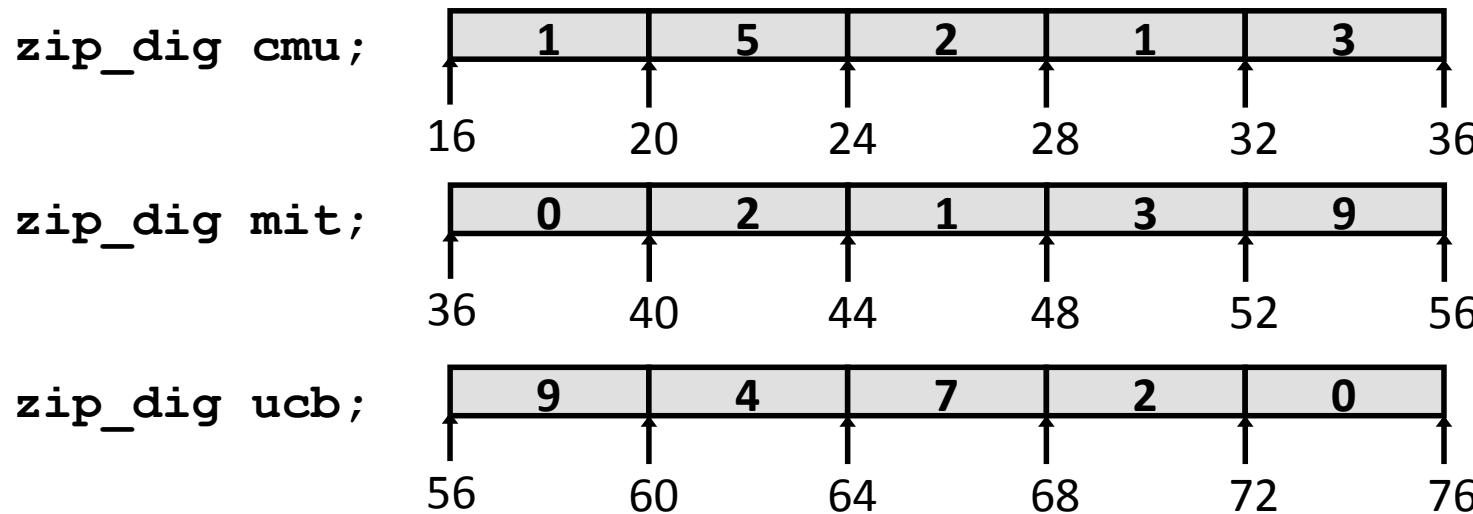
| Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	$x + 4$
`&val[2]`	`int *`	$x + 8$
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + `i	`int *`	$x + 4\,i$

Array Example
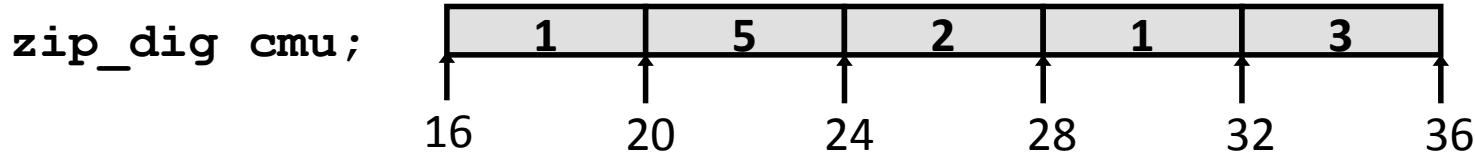
```
typedef int zip_dig[5];

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

`zip_dig cmu;`

1	5	2	1	3

16   20   24   28   32   36

`zip_dig mit;`

0	2	1	3	9

36   40   44   48   52   56

`zip_dig ucb;`

9	4	7	2	0

56   60   64   68   72   76

- **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;`

| 1	5	2	1	3

16   20   24   28   32   36

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```
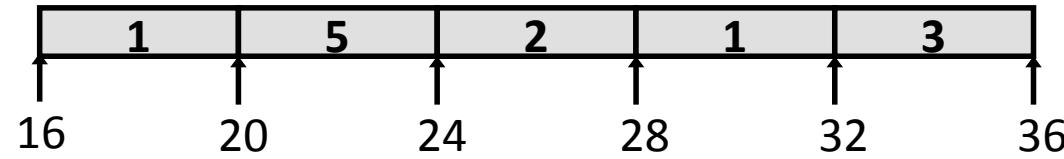
IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```
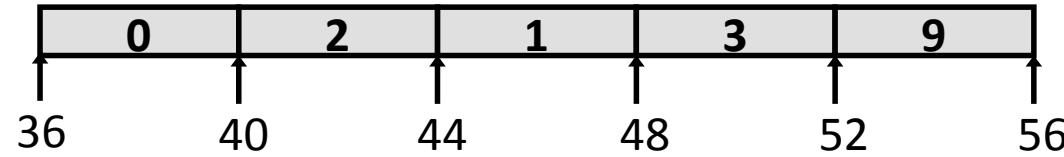
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

Referencing Examples
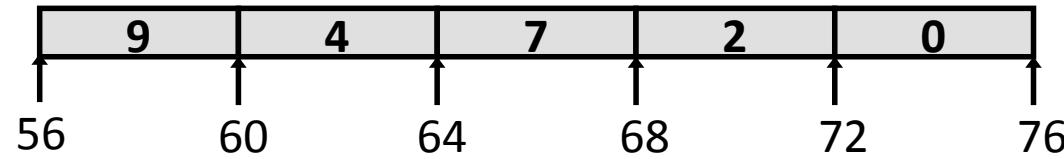
`zip_dig cmu;`

1	5	2	1	3

16　　　20　　　24　　　28　　　32　　　36

`zip_dig mit;`

0	2	1	3	9

36　　　40　　　44　　　48　　　52　　　56

`zip_dig ucb;`

9	4	7	2	0

56　　　60　　　64　　　68　　　72　　　76

■ **Reference**　　　**Address**　　　　　　**Value**　　　**Guaranteed?**
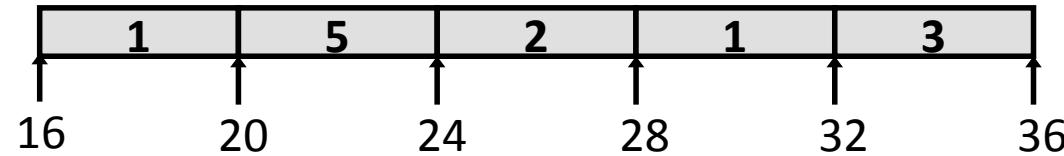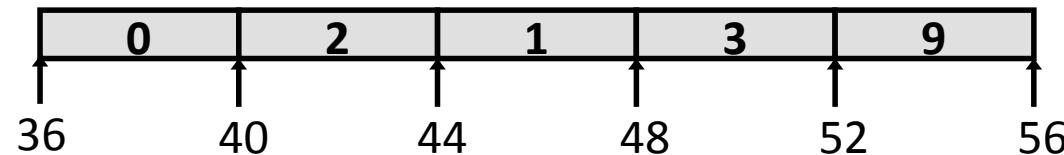
`mit[3]`

`mit[5]`

`mit[-1]`

`cmu[15]`

Will disappear Blackboard?

Referencing Examples

```
zip_dig cmu;
```

1	5	2	1	3

16   20   24   28   32   36

```
zip_dig mit;
```

0	2	1	3	9

36   40   44   48   52   56

```
zip_dig ucb;
```

9	4	7	2	0

56   60   64   68   72   76
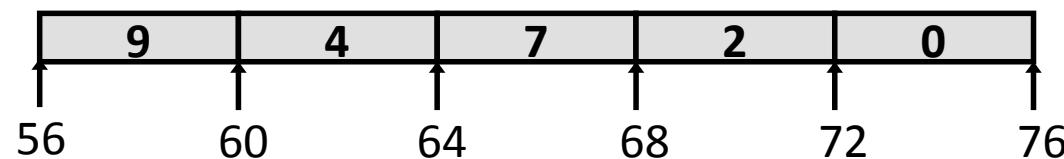
- **Reference**

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)

```c
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
  ```
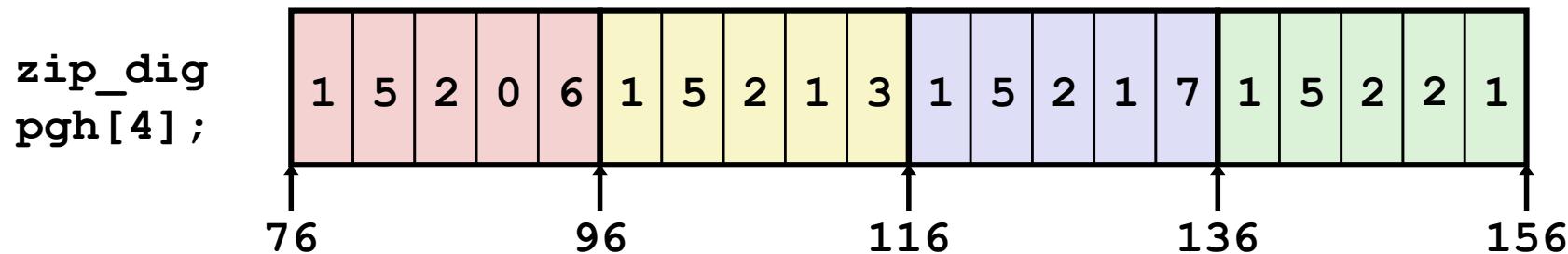
- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
  - `z++` increments by 4

```c
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];`

| 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

76   96   116   136   156

- **"`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 $\mathtt{A}[R][C];$

  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes
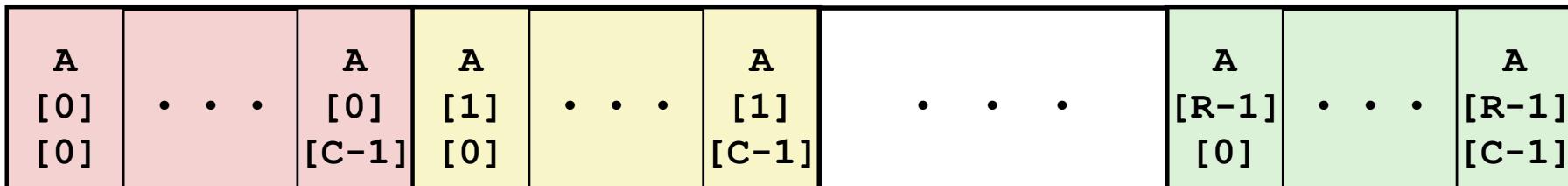
- **Array Size**

  - $R * C * K$ bytes

- **Arrangement**

  - Row-Major Ordering

$$\begin{bmatrix} \mathtt{A[0][0]} & \cdots & \mathtt{A[0][C-1]} \\ & \vdots & \\ \mathtt{A[R-1][0]} & \cdots & \mathtt{A[R-1][C-1]} \end{bmatrix}$$

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

A [0] [0]	· · ·	A [0] [C-1]	A [1] [0]	· · ·	A [1] [C-1]	· · ·	A [R-1] [0]	· · ·	A [R-1] [C-1]

$\longleftarrow$ **4*R*C** Bytes $\longrightarrow$
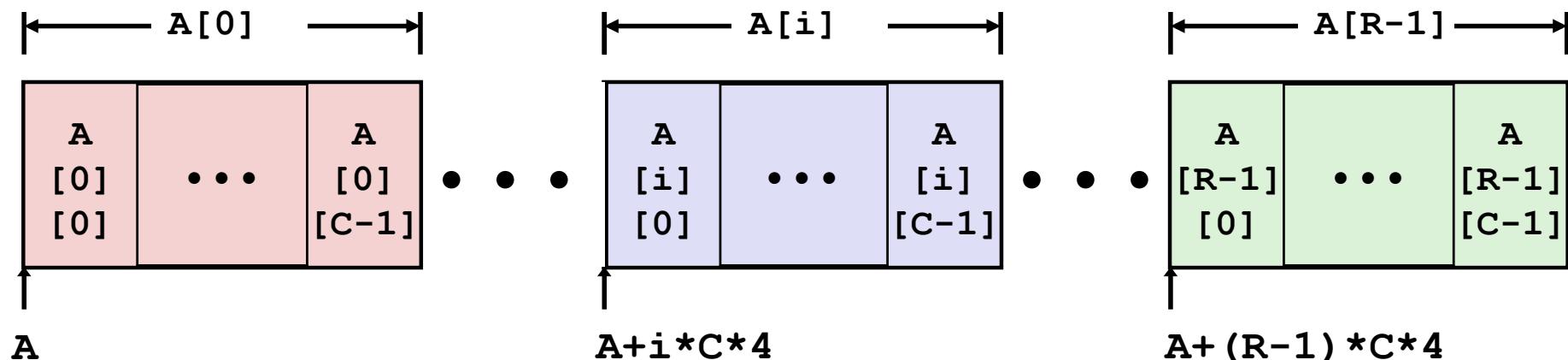
Nested Array Row Access

- **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)`
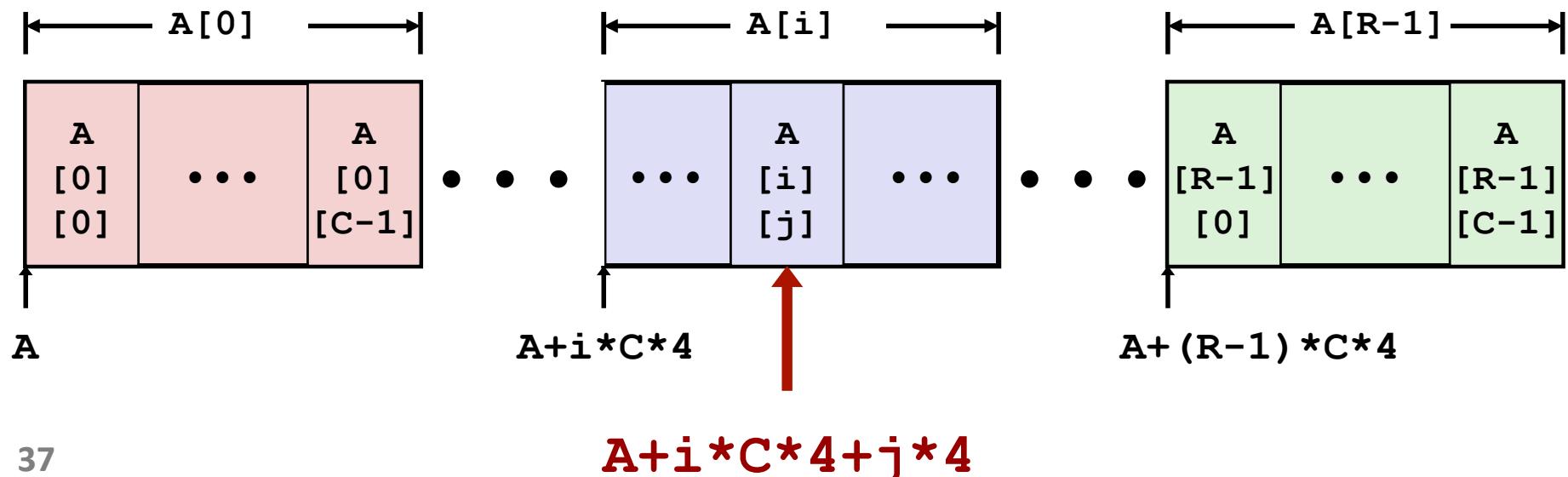
Nested Array Row Access

- **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];`

$\overset{\longleftarrow\;A[0]\;\longrightarrow}{}$				$\overset{\longleftarrow\;A[i]\;\longrightarrow}{}$				$\overset{\longleftarrow\;A[R-1]\;\longrightarrow}{}$	

A
[0]
[0] ••• A
[0]
[C-1] • • • ••• A
[i]
[j] ••• • • • A
[R-1]
[0] ••• A
[R-1]
[C-1]

`A` `A+i*C*4` `A+(R-1)*C*4`

`A+i*C*4+j*4`

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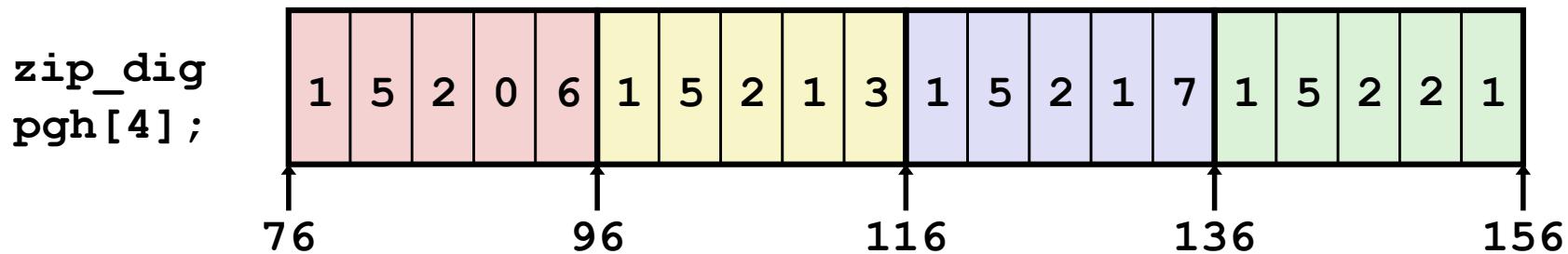
- **Array Elements**
  - `pgh[index][dig]` is `int`
  - Address: `pgh + 20*index + 4*dig`
- **IA32 Code**
 - Computes address `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference
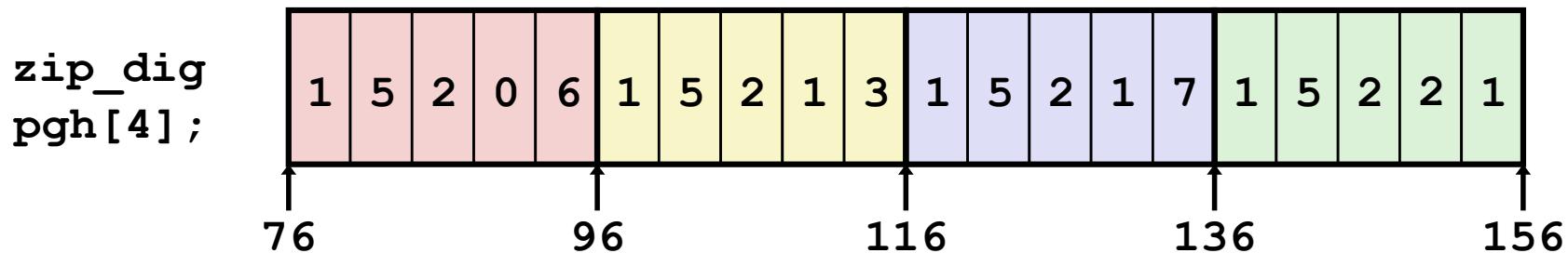
Strange Referencing Examples

zip_dig pgh[4];

| | 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

76   96   116   136   156

- **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]; | 1 | 5 | 2 | 0 | 6 | 1 | 5 | 2 | 1 | 3 | 1 | 5 | 2 | 1 | 7 | 1 | 5 | 2 | 2 | 1 |

```
76              96             116            136            156
```

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

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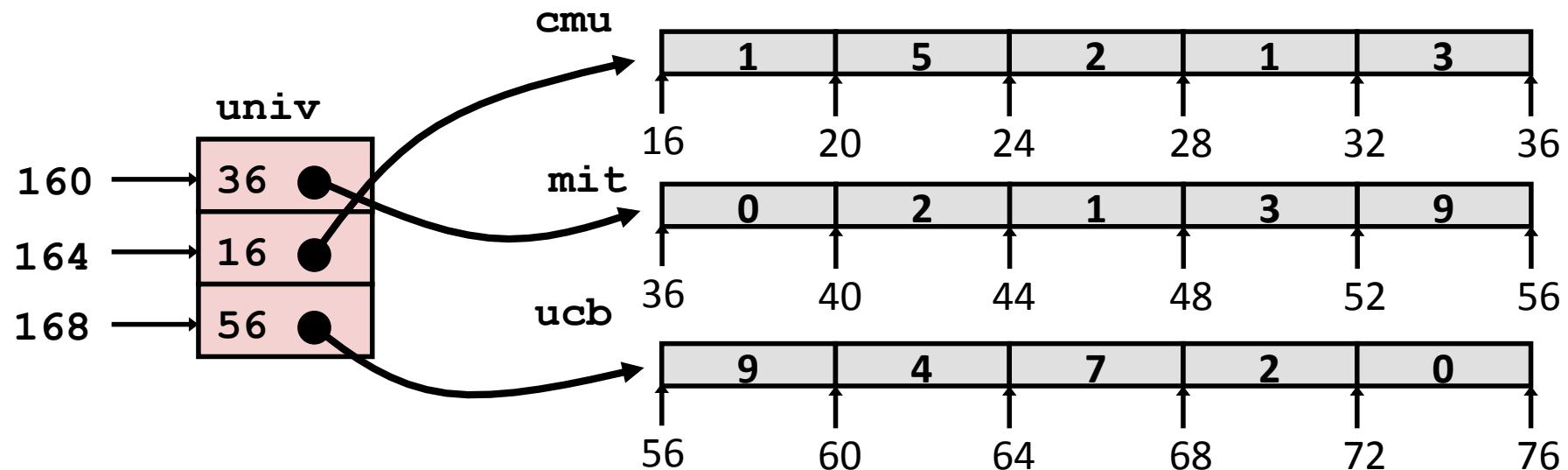
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

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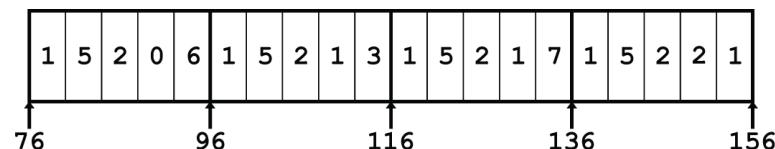
- **Computation (IA32)**
  - Element access **Mem[Mem[univ+4*index]+4*dig]**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Array Element Accesses
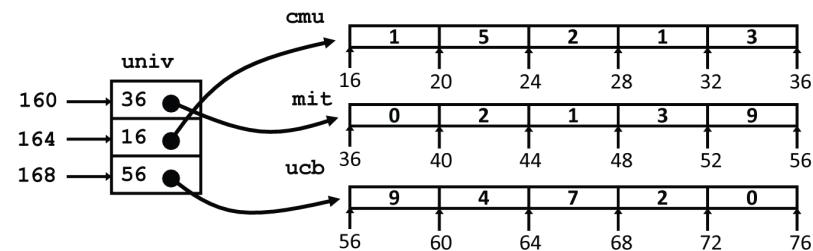
Nested array

```
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];
}
```
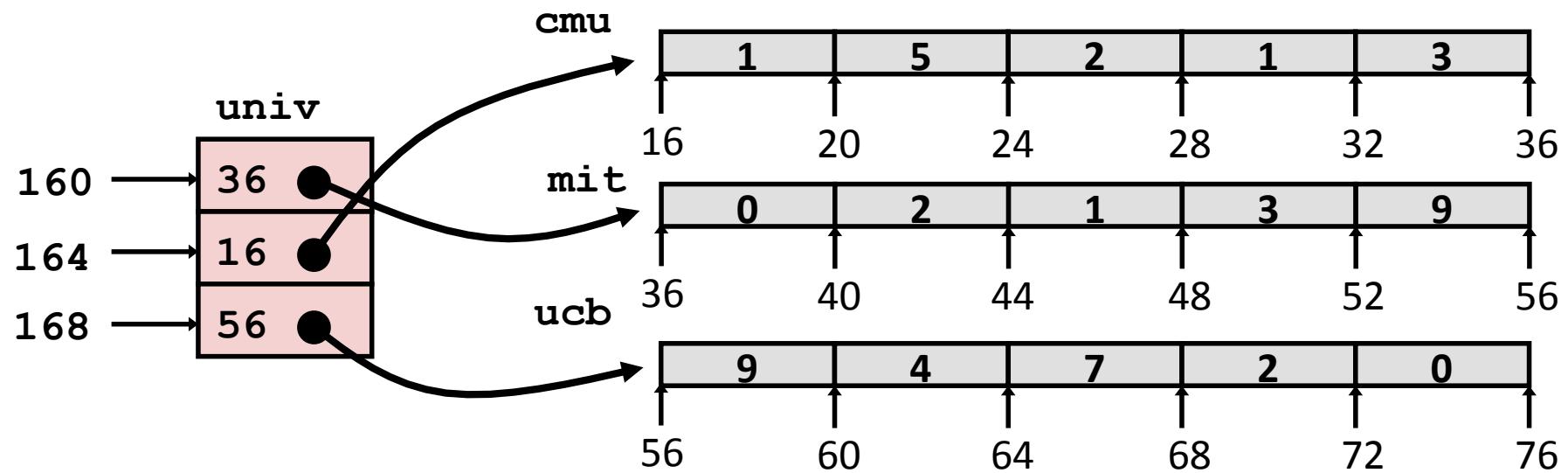

Access looks similar, but element:

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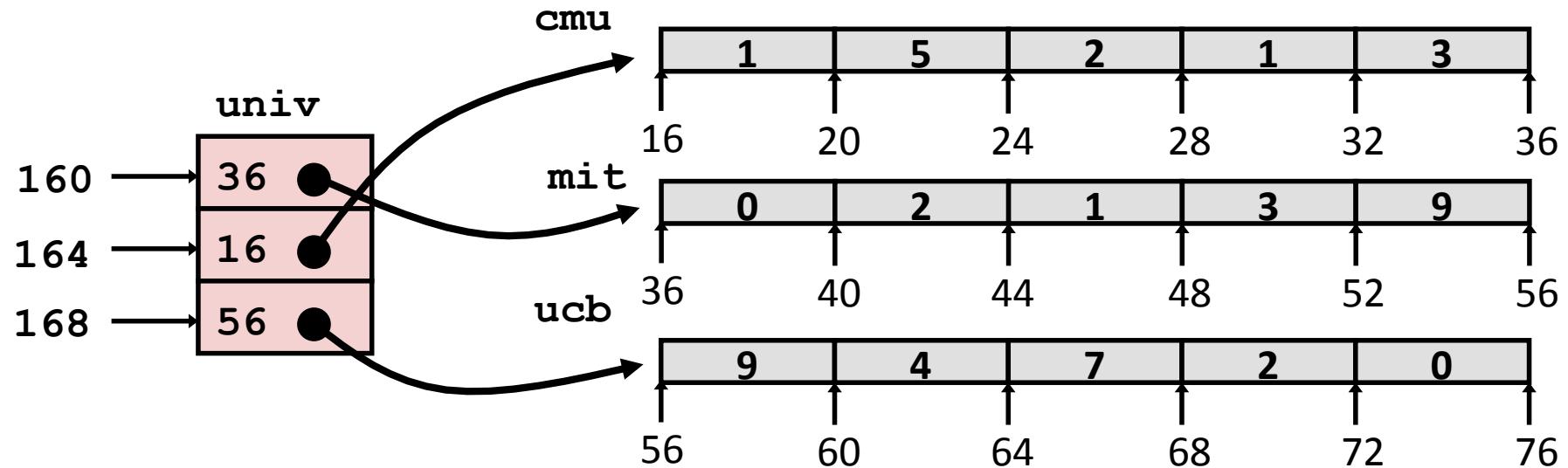
`Mem[Mem[univ+4*index]+4*dig]`

Strange Referencing Examples

cmu

1	5	2	1	3

16   20   24   28   32   36

univ

160 → | 36 ● |
164 → | 16 ● |
168 → | 56 ● |

mit

0	2	1	3	9

36   40   44   48   52   56

ucb

9	4	7	2	0

56   60   64   68   72   76

■ **Reference**   **Address**   **Value**   **Guaranteed?**

`univ[2][3]`

`univ[1][5]`

`univ[2][-1]`   **Will disappear**

`univ[3][-1]`

`univ[1][12]`

Strange Referencing Examples

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

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
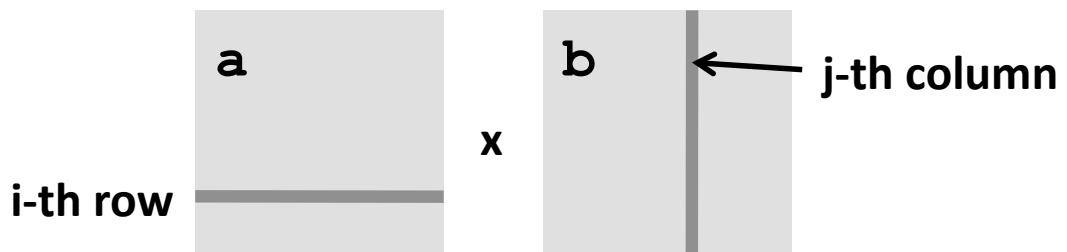
- **Strengths**
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
  - Avoids multiply in index computation

- **Limitation**
  - Only works for fixed array size

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

```c
/* 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;
}
```


a   x   b   ← j-th column

i-th row

Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size

- **Programming**
  - Must do index computation explicitly

- **Performance**
  - Accessing single element costly
  - Must do multiplication

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

```c
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

```c
/* 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;
}
```
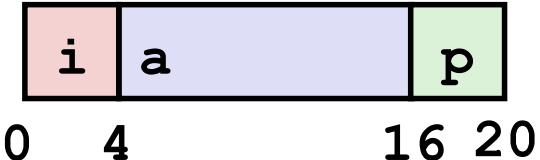
Today

- **Procedures (x86-64)**
- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**

Structures

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

Memory Layout

i	a	p
0 4		16 20

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

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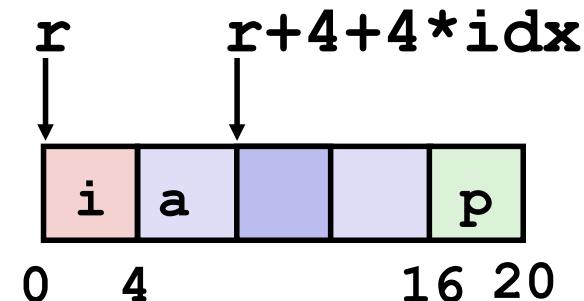
```
void
set_i(struct rec *r,
      int val)
{
    r->i = val;
}
```

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;
};
```

r   r+4+4*idx

i	a			p

0   4   16 20

- **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];
}
```


Element 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
```