IA32 Stack Discipline From Last Time

- **Stack grows down, high addresses to low**
- **%esp points to lowest allocated position on stack**
- **Pushl**
  - **%esp-=4 , write word to memory %esp points to**
- **Popl**
  - **Read word from memory %esp points to, %esp+=4**
- **Call instruction**
  - **Pushes %eip (pointer to next instruction)**
  - **Jumps to target**
- **Ret**
  - **Pops into %eip (returns to next next instruction after call)**
- **Stack "frame" stores the context in which the procedure operates**
- **Stack-based languages**
  - **Stack stores context of procedure calls**
  - **Multiple calls to a procedure can be outstanding simultaneously**
  - **Recursion**
  - **Sorry attempt to connect to modern French philosophy**

Call Chain Example

Code Structure

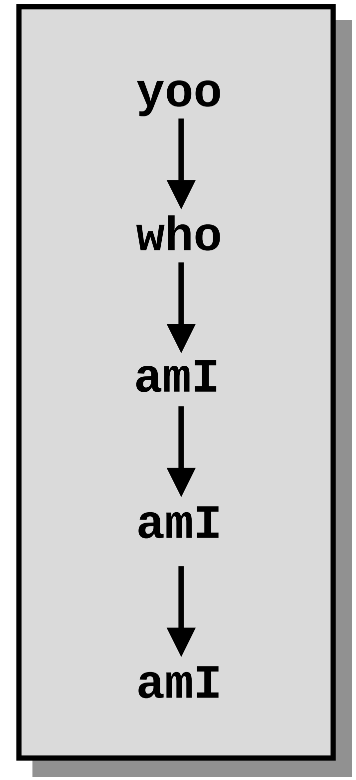
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   •
   •
   amI();
   •
   •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

- Procedure `amI` recursive

Call Chain



IA32 Stack Structure

Stack Growth

- Toward lower addresses

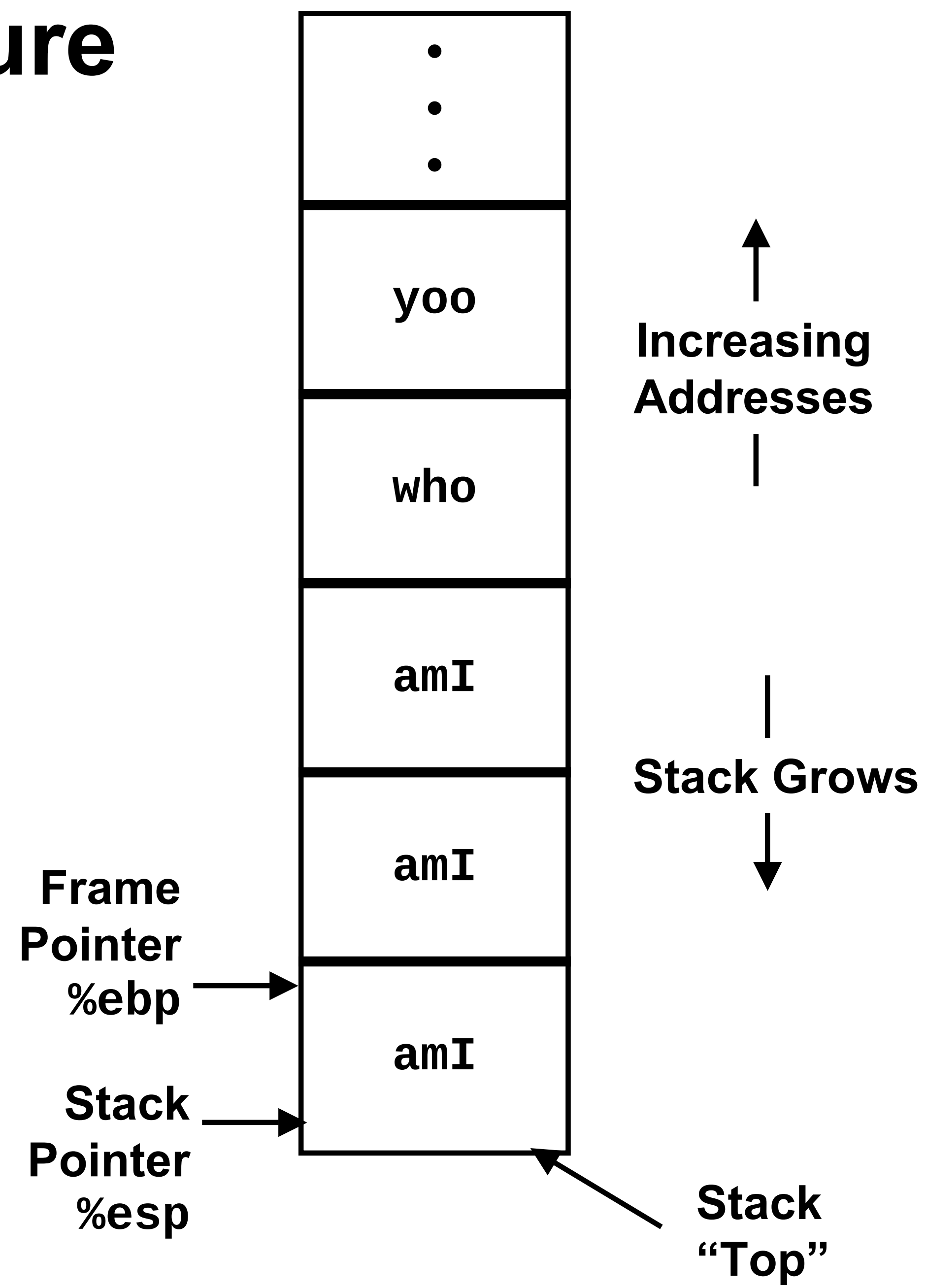
Stack Pointer

- Address of highest allocated item in stack
- Use register %esp

Frame Pointer

- Start of current stack frame
- Use register %ebp

Procedure Call Conventions



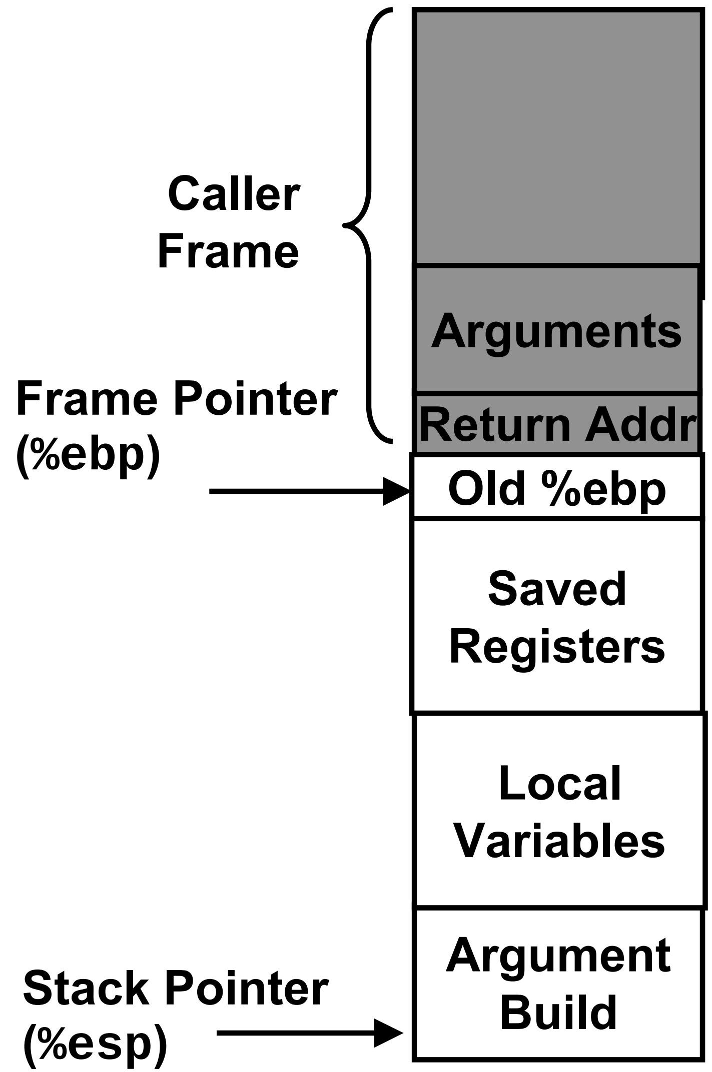
IA32/Linux Stack Frame

Caller Stack Frame

- **Arguments for this call**
  - Pushed explicitly
- **Return address**
  - Pushed by `call` instruction

Callee Stack Frame

- **Old frame pointer**
- **Saved register context**
- **Local variables**
  - If can't keep in registers
- **Parameters for called functions**

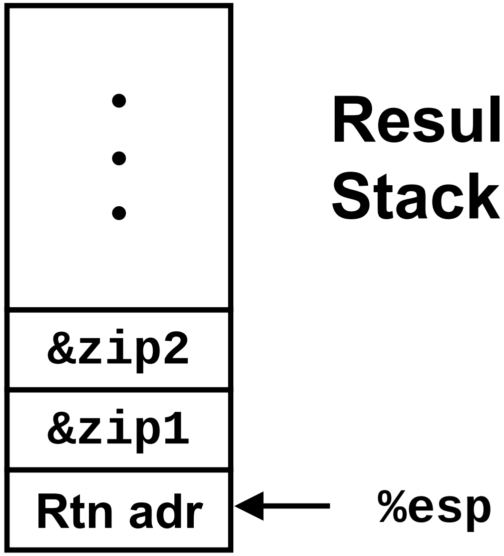


Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
call_swap:
   • • •
   pushl $zip2
   pushl $zip1
   call swap
   • • •
```

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



Resulting Stack

Revisiting swap

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

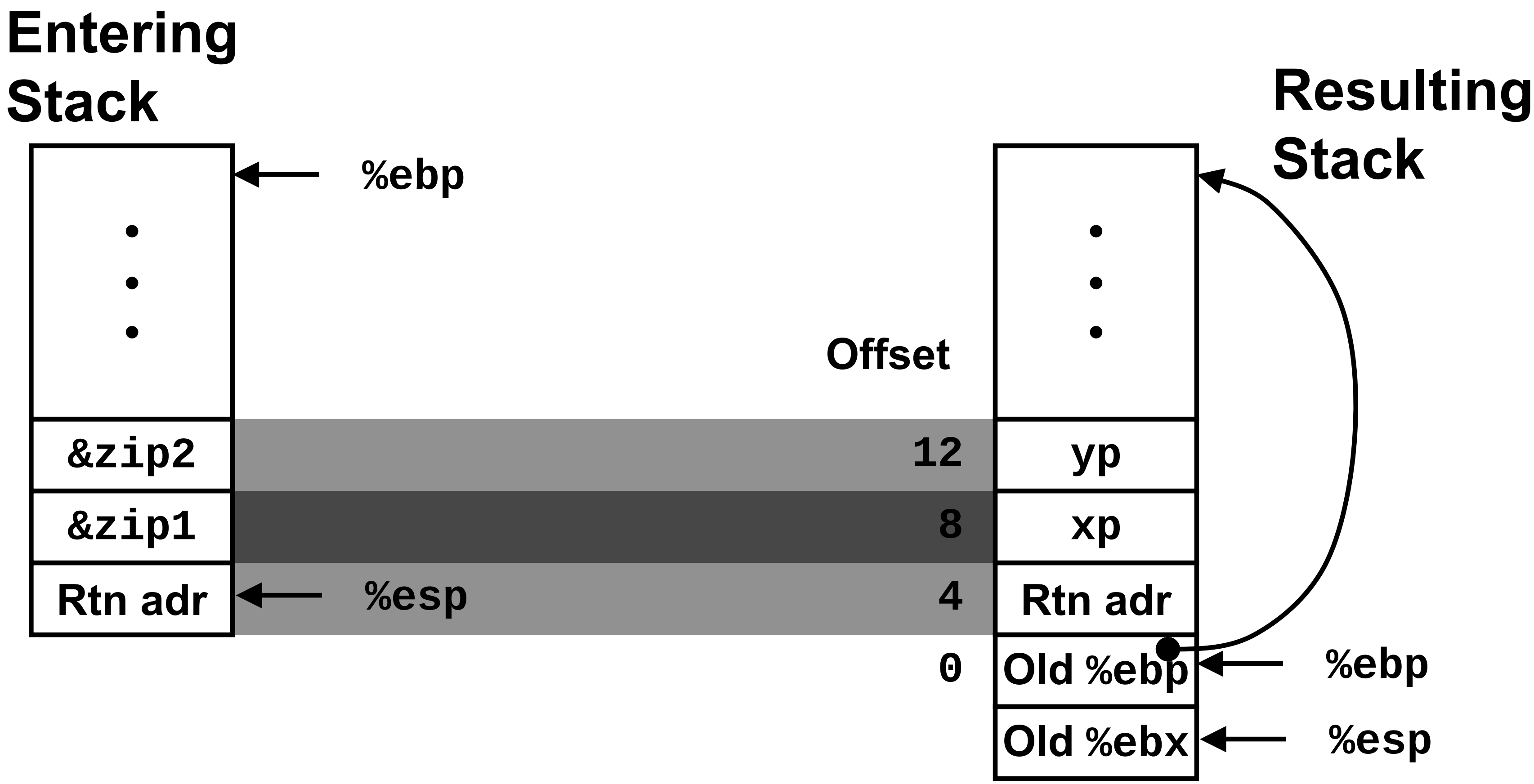
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up (pushl %ebp … pushl %ebx)

Body (movl 12(%ebp),%ecx … movl %ebx,(%ecx))

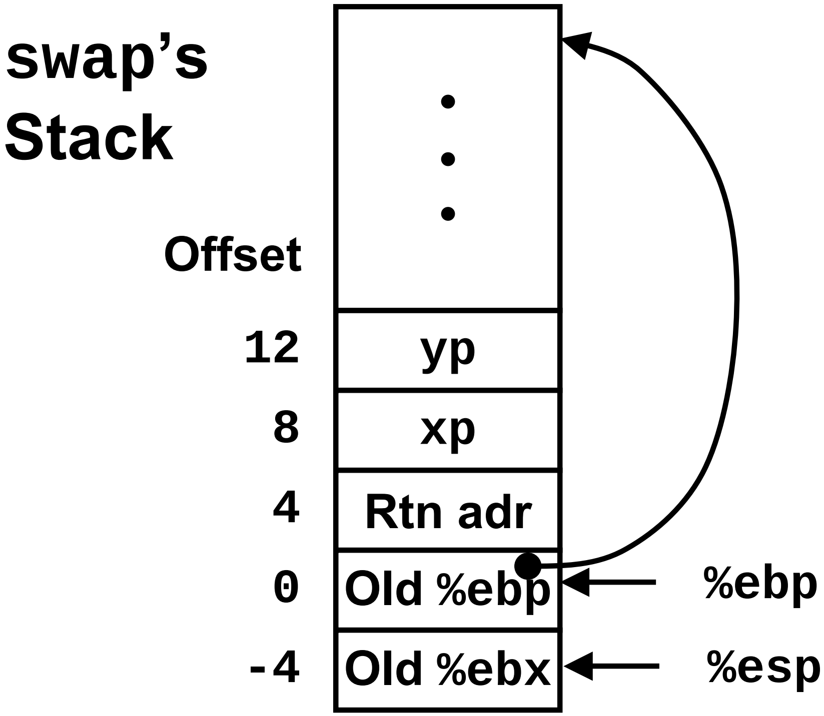
Finish (movl -4(%ebp),%ebx … ret)

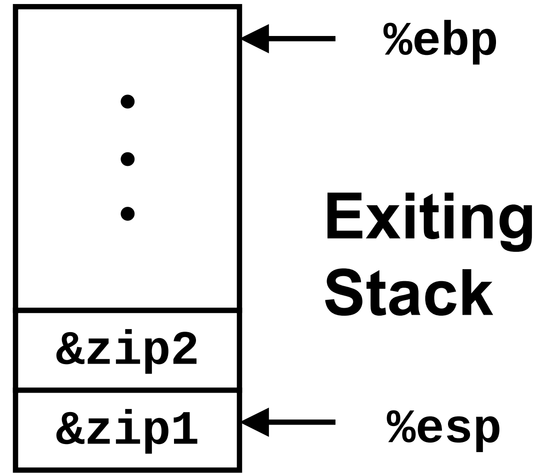
swap Setup




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Finish






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register %ebx
- Didn't do so for %eax, %ecx, or %edx

Register Saving Conventions

When procedure yoo calls who:

- yoo is the *caller*, who is the *callee*

Can Register be Used for Temporary Storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- Contents of register %edx overwritten by who

Conventions

- **"Caller Save"**
  - Caller saves temporary in its frame before calling
- **"Callee Save"**
  - Callee saves temporary in its frame before using

IA32/Linux Register Usage

- **Surmised by looking at code examples**

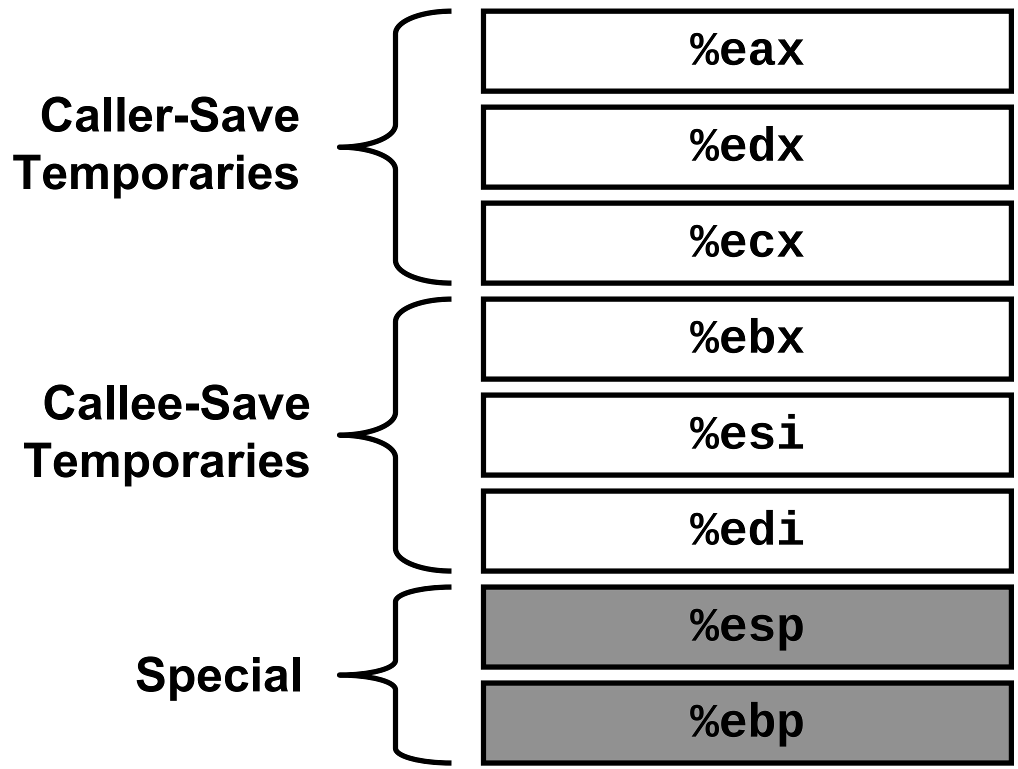
Integer Registers

- **Two have special uses**

  %ebp, %esp
- **Three managed as callee-save**

  %ebx, %esi, %edi
  - Old values saved on stack prior to using
- **Three managed as caller-save**

  %eax, %edx, %ecx
  - Do what you please, but expect any callee to do so, as well
- **Register %eax also stores returned value**



Recursive Factorial

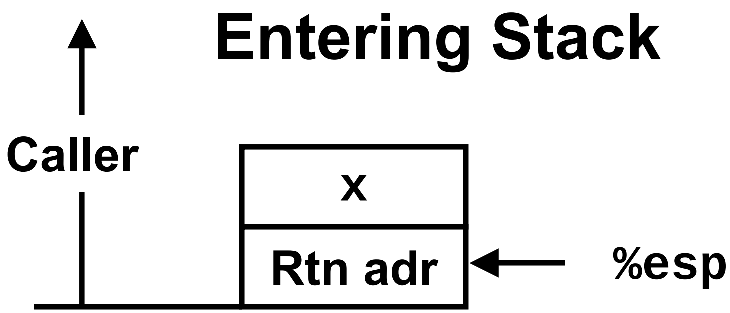
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

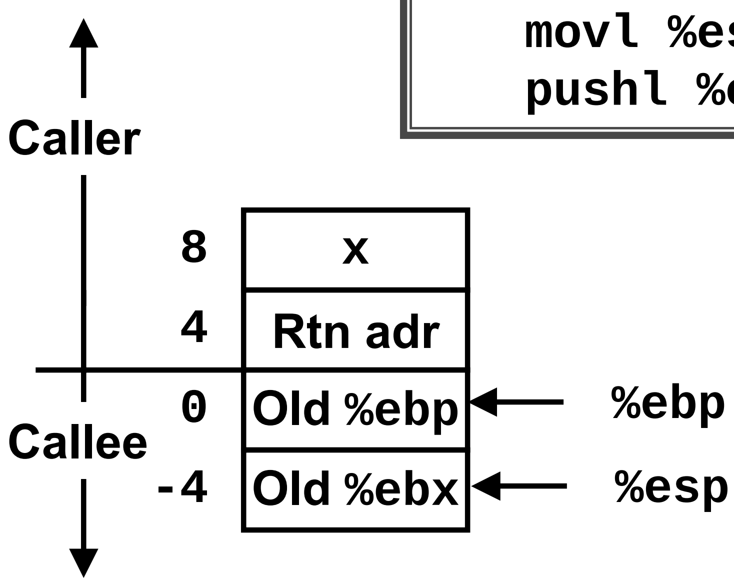
Complete Assembly

- **Assembler directives**
  - Lines beginning with "`.`"
  - Not of concern to us
- **Labels**
  - `.Lxx`
- **Actual instructions**

Rfact Stack Setup




```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Rfact Body

```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:                # Term:
  movl $1,%eax         # return val = 1
.L79:                # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Registers

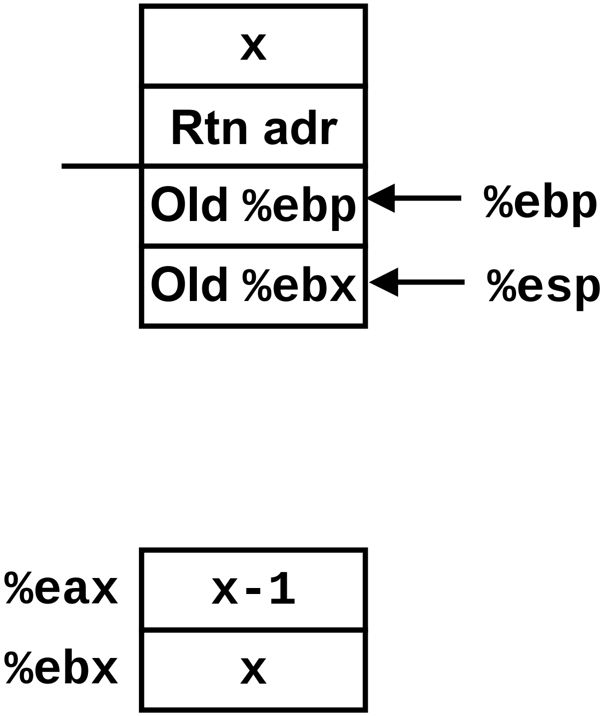
`$ebx` Stored value of x

`$eax`

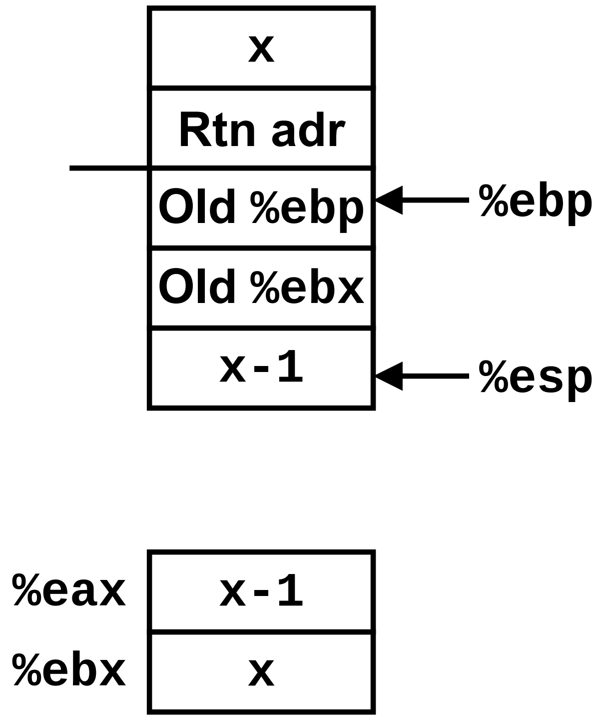
- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

Rfact Recursion

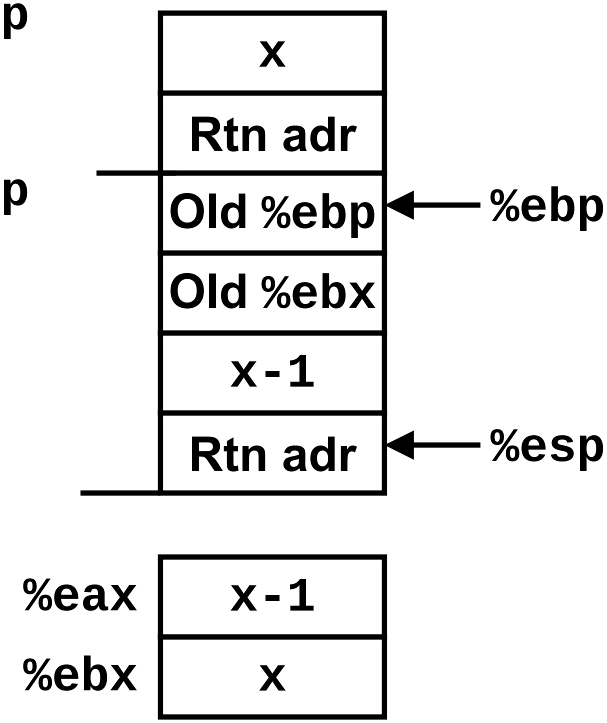
```
leal -1(%ebx),%eax
```



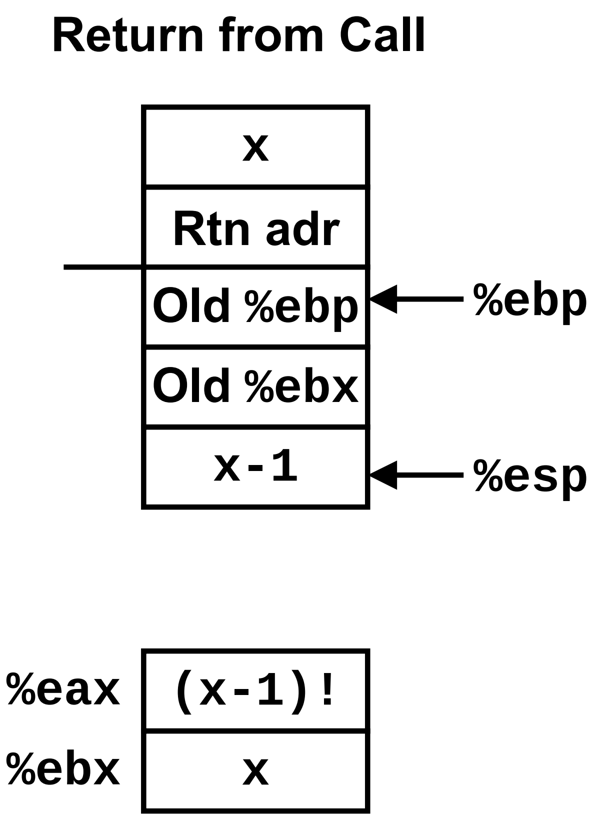

```
pushl %eax
```

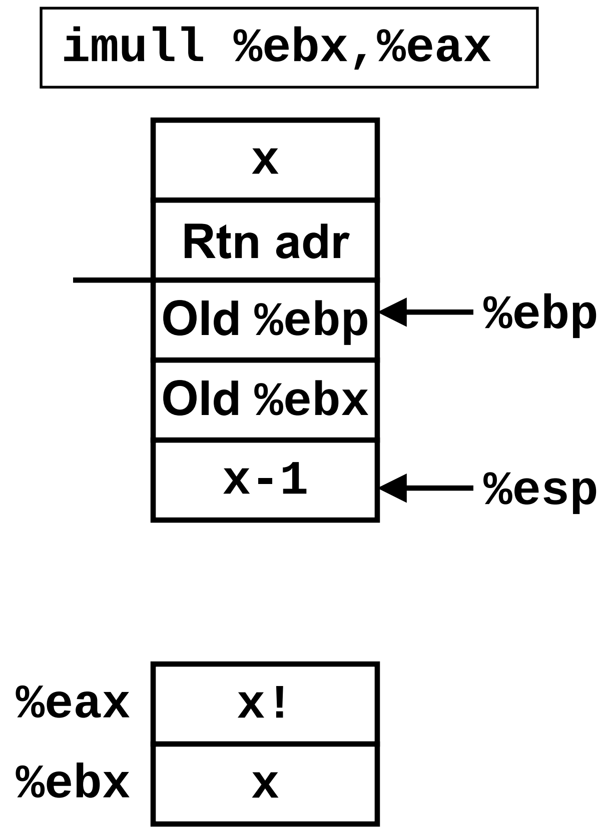



```
call rfact
```

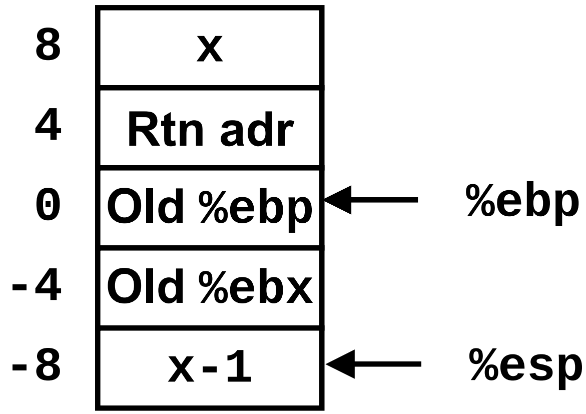


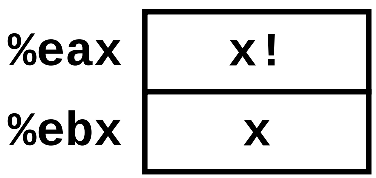
Rfact Result



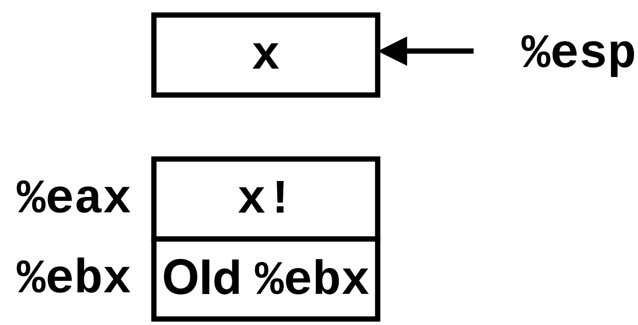


Rfact Completion






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```



Tail Recursion and Optimization

- **Tail recursive procedures can be turned into iterative procedures (for loops)**
- **Compilers can sometimes detect tail recursion and do the conversion for you**

```
void tail_rec(…) {
    …
    tail_rec(…);
}
```

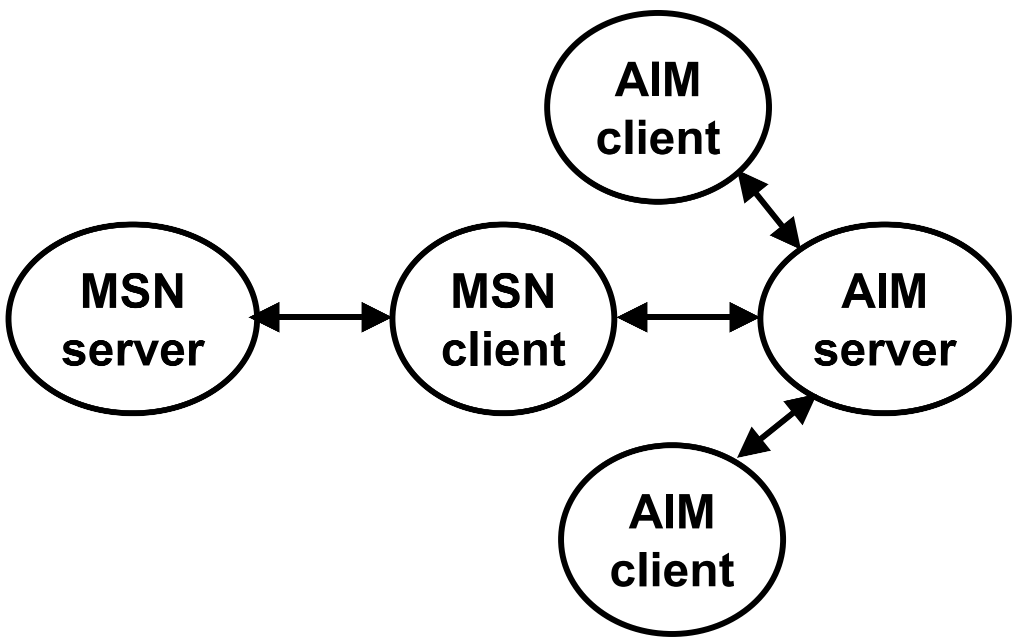
Internet worm and IM War

November, 1988

- Internet Worm attacks thousands of Internet hosts.
- How did it happen?

July, 1999

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



Internet Worm and IM War (cont)

August 1999

- **Mysteriously, Messenger clients can no longer access AIM servers.**
- **Even though the AIM protocol is an open, published standard.**
- **Microsoft and AOL begin the IM war:**
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes.
  - At least 13 such skirmishes.
- **How did it happen?**

The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!

- many Unix functions, such as gets() and strcpy(), do not check argument sizes.
- allows target buffers to overflow.

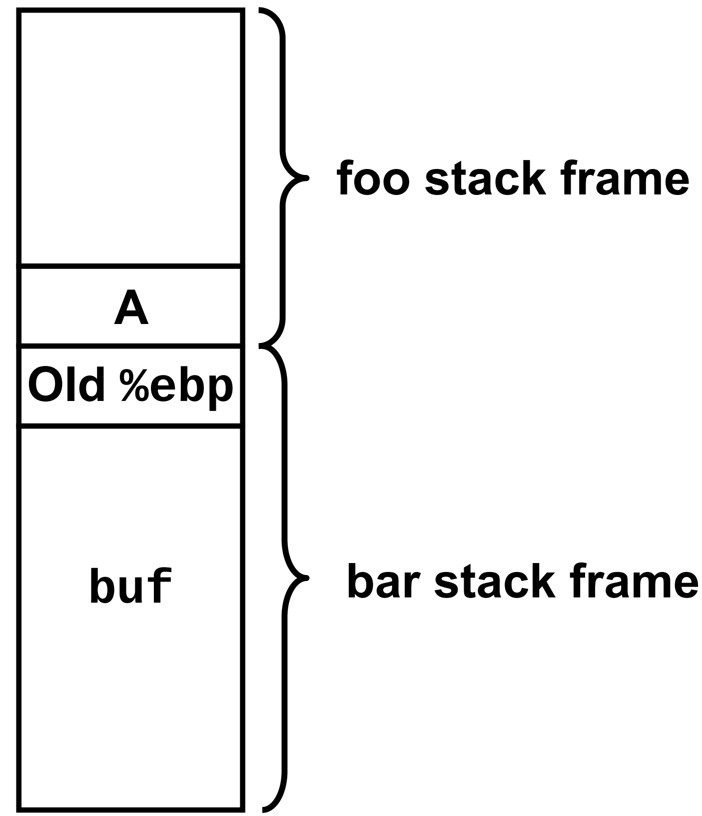
Stack buffer overflows

return address A → (points to the line after `bar();`)

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```

Stack before call to `gets()`



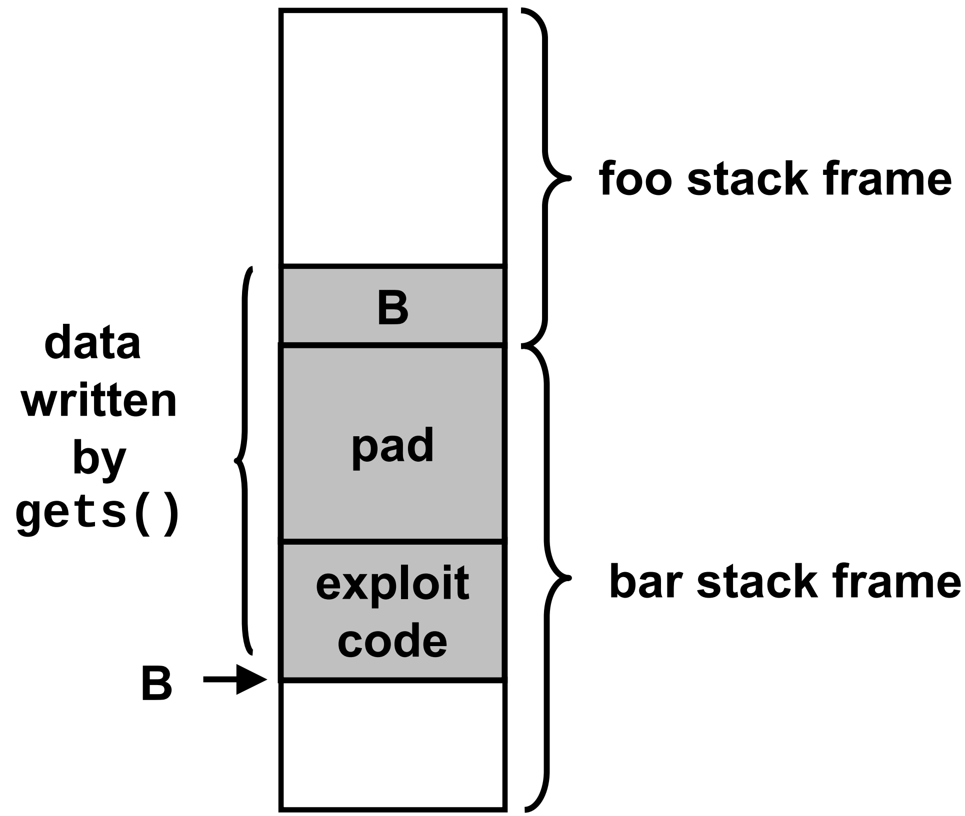
Stack buffer overflows (cont)

return address A →

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```

Stack after call to gets()



When bar() returns, control passes silently to B instead of A!!

Exploits often based on buffer overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.

Internet worm

- **Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:**
  - *`finger pdinda@cs.northwestern.edu`*
- **Worm attacked fingerd client by sending phony argument:**
  - *`finger “exploit code  padding  new return address”`*
  - **exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.**

IM War

- **AOL exploited existing buffer overflow bug in AIM clients**
- **exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.**
- **When Microsoft changed code to match signature, AOL changed signature location.**

Main Ideas

Stack Provides Storage for Procedure Instantiation

- Save state
- Local variables
- Any variable for which must create pointer

Assembly Code Must Manage Stack

- Allocate / deallocate by decrementing / incrementing stack pointer
- Saving / restoring register state

Stack Adequate for All Forms of Recursion

- Including multi-way and mutual recursion examples in the bonus slides.

Good programmers know the stack discipline and are aware of the dangers of stack buffer overflows.

`And now… structured data…`

Basic Data Types

Integral

- Stored & operated on in general 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`

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	`long double`

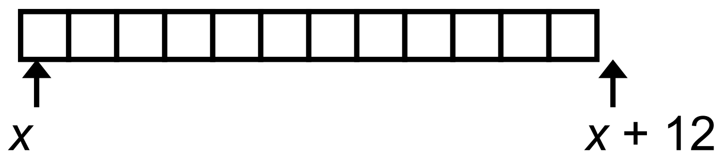
Array Allocation

Basic Principle

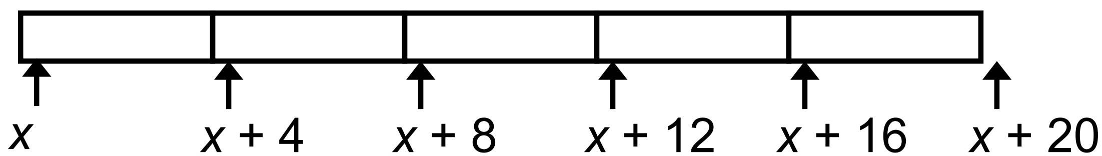
T **A[*L*];**

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

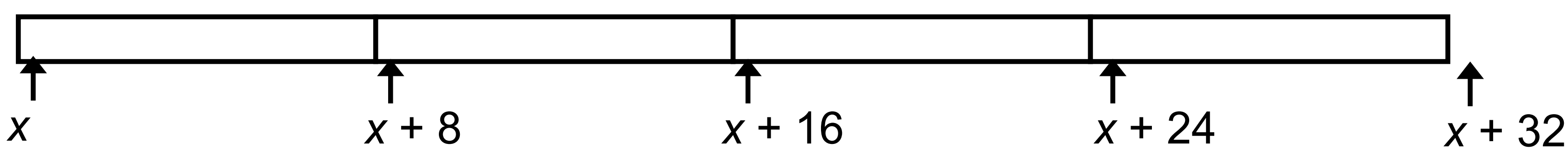
`char string[12];`



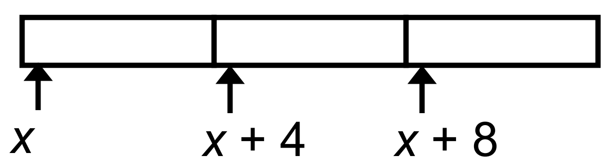
`int val[5];`



`double a[4];`



`char *p[3];`



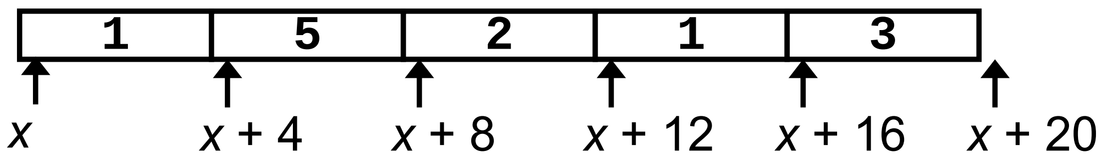
Array Access

Basic Principle

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

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

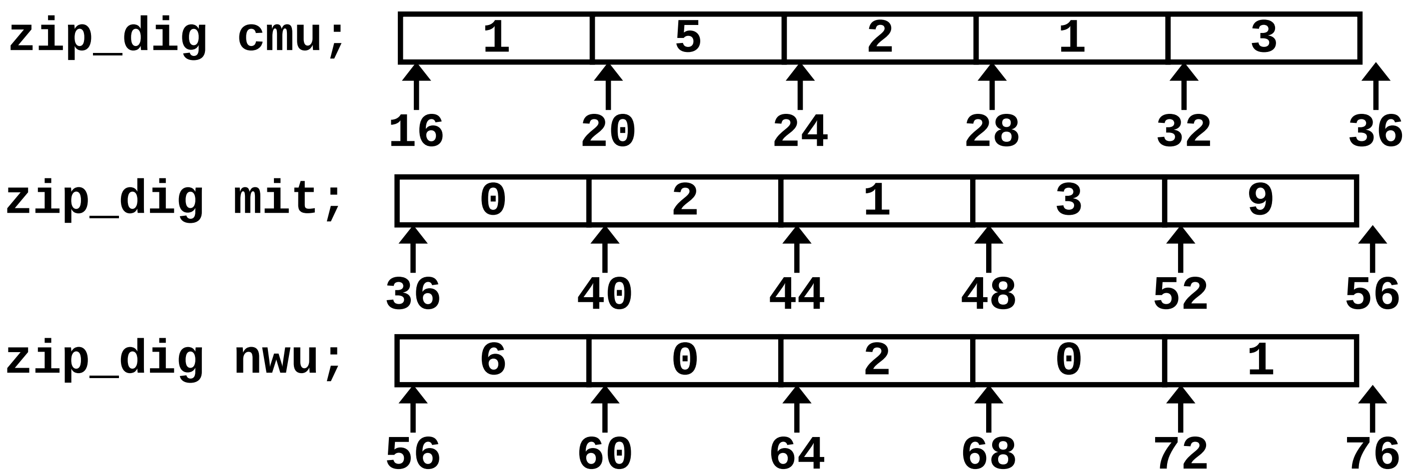
```
int val[5];
```



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 nwu = { 6, 0, 2, 0, 1 };
```



Notes

- **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

Computation

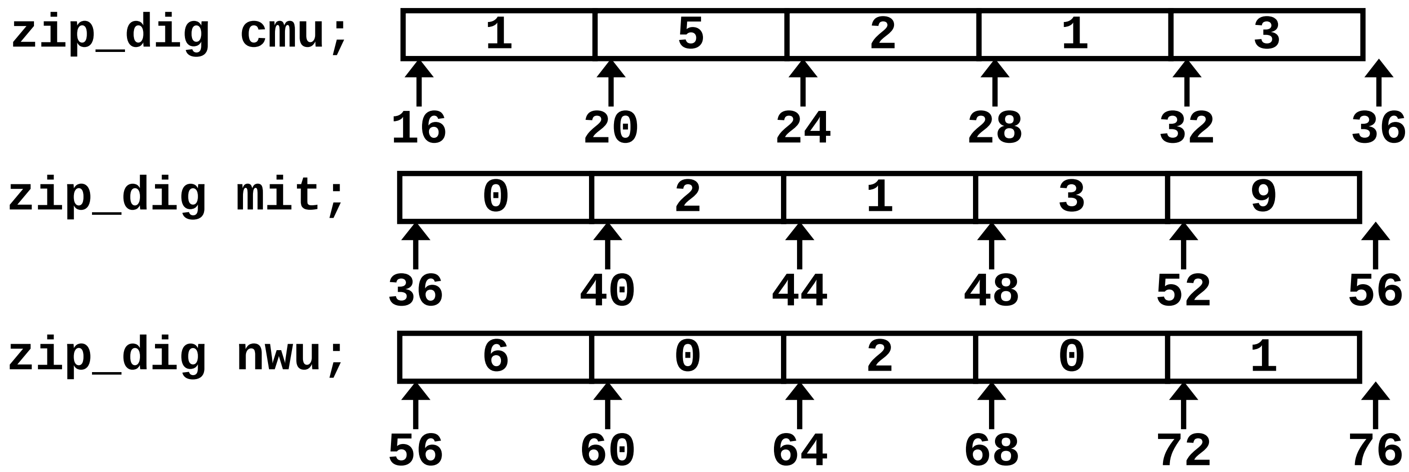
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

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

Memory Reference Code

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

Referencing Examples



Code Does Not Do Any Bounds Checking!

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

- **Out of range behavior implementation-dependent**
  - No guranteed relative allocation of different arrays

Array Loop Example

Original Source

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

Transformed Version

- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form
  - No need to 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

Registers

```
%ecx   z
%eax   zi
%ebx   zend
```

Computations

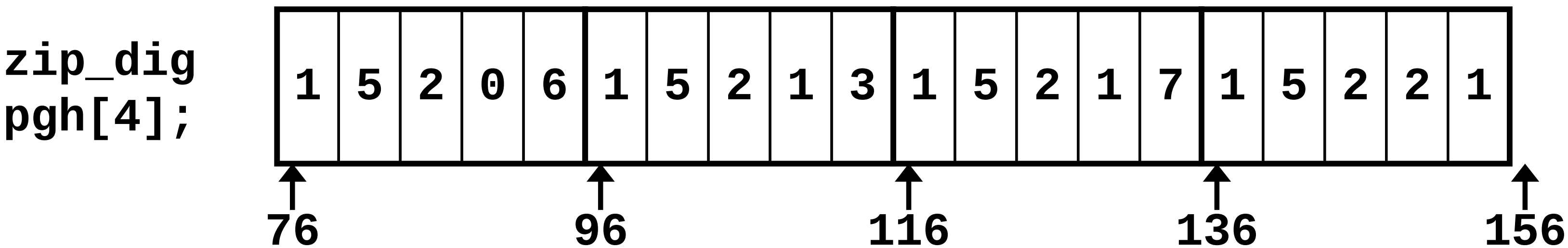
- `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
- `z++` increments by 4

```
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 }};
```



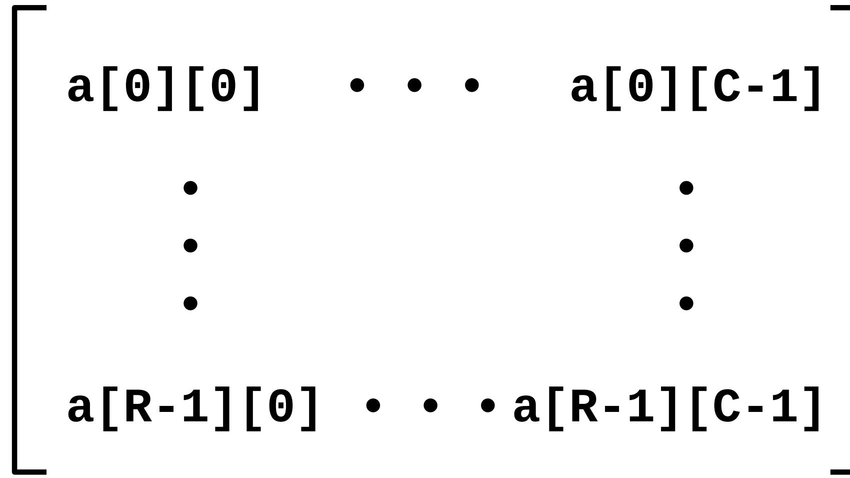
- **Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **pgh** denotes array of 4 elements
    - » Allocated contiguously
  - Each element is an array of 5 **`int`**'s
    - » Allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

Nested Array Allocation

Declaration

T **A[*R*][*C*];**

- **Array of data type *T***
- ***R* rows**
- ***C* columns**
- **Type *T* element requires *K* bytes**



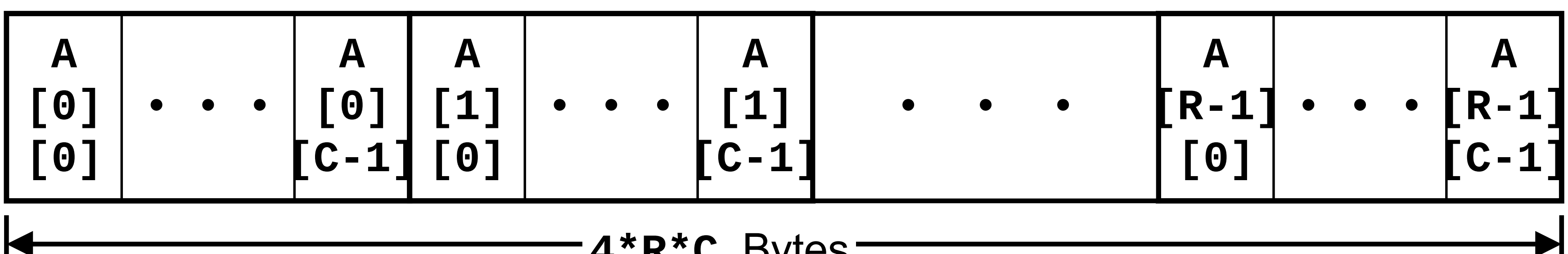
Array Size

- *R* * *C* * *K* **bytes**

Arrangement

- Row-Major Ordering

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

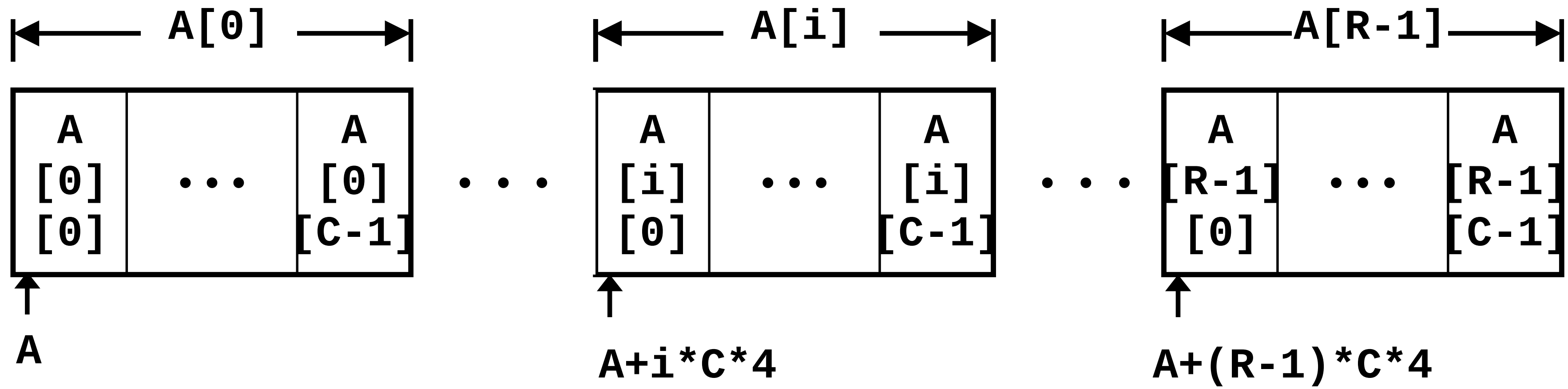


Nested Array Row Access

Row Vectors

- `A[i]` is array of *C* elements
- Each element of type *T*
- Starting address `A +` *i* * *C* * *K*

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



Nested Array Row Access Code

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

Row Vector

- pgh[index] is array of 5 int's
- Starting address pgh+20*index

Code

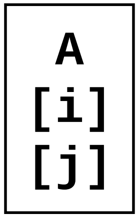
- Computes and returns address
- Compute as pgh + 4*(index+4*index)

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

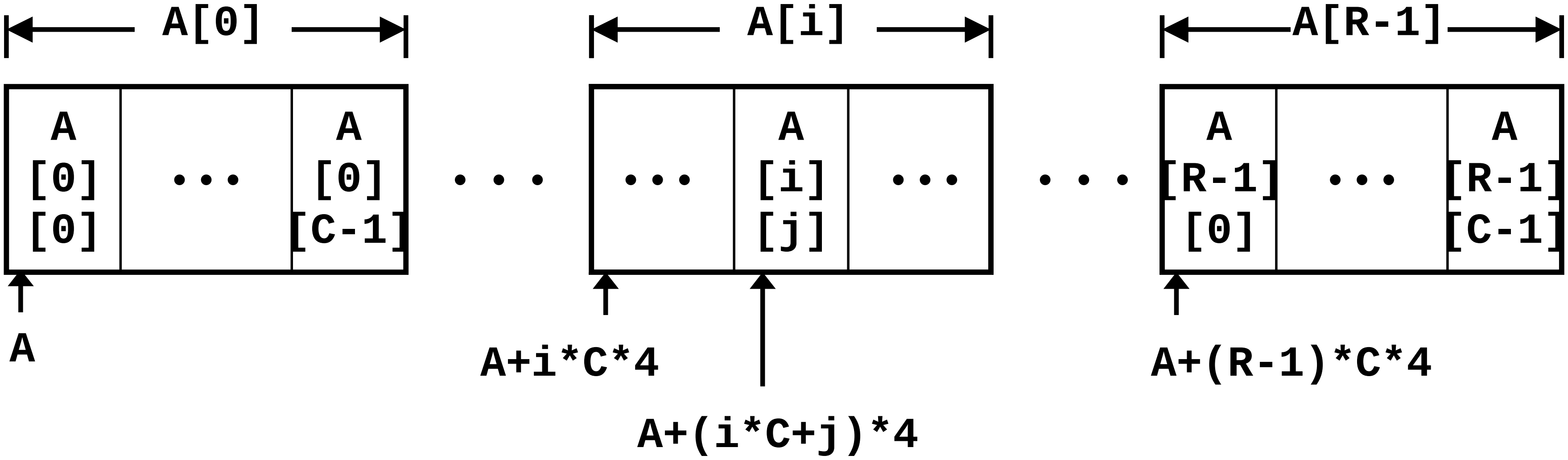
Nested Array Element Access

Array Elements

- `A[i][j]` is element of type *T*
- Address `A +` (*i* * *C* + *j*) * *K*



`int A[R][C];`



Nested Array Element Access Code

Array Elements

- `pgh[index][dig]` is `int`
- Address:

  `pgh + 20*index + 4*dig`

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

Code

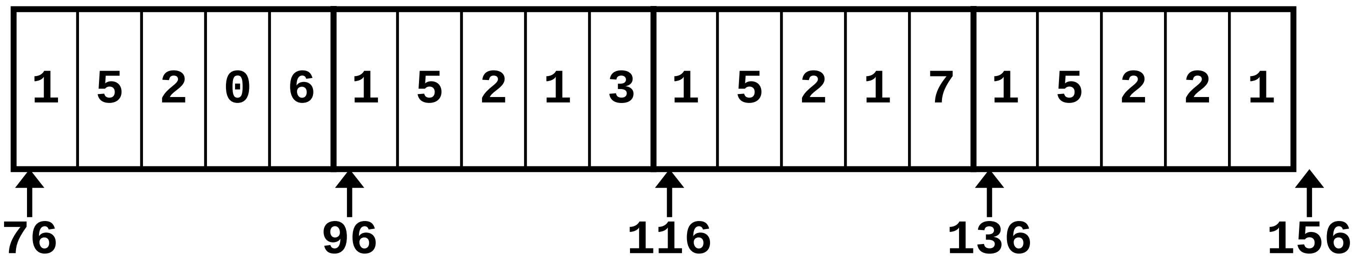
- Computes address

  `pgh + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference

```
# %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)
```

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

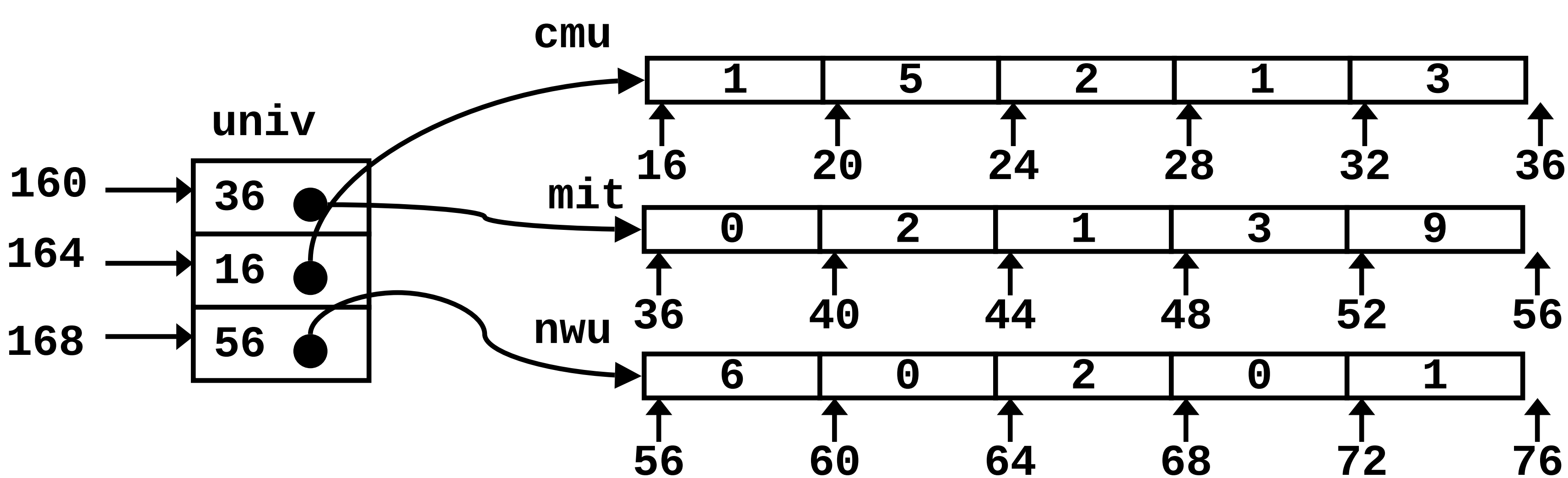
- **Code does not do any bounds checking**
- **Ordering of elements within array guaranteed**

Multi-Level Array Example

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

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig nwu = { 6, 0, 2, 0, 1 };
```

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



Referencing “Row” in Multi-Level Array

Row Vector

- `univ[index]` is pointer to array of `int`’s
- Starting address `Mem[univ+4*index]`

```
int* get_univ_zip(int index)
{
  return univ[index];
}
```

Code

- Computes address within `univ`
- Reads pointer from memory and returns it

```
# %edx = index
leal 0(,%edx,4),%eax   # 4*index
movl univ(%eax),%eax   # *(univ+4*index)
```

Accessing Element in Multi-Level Array

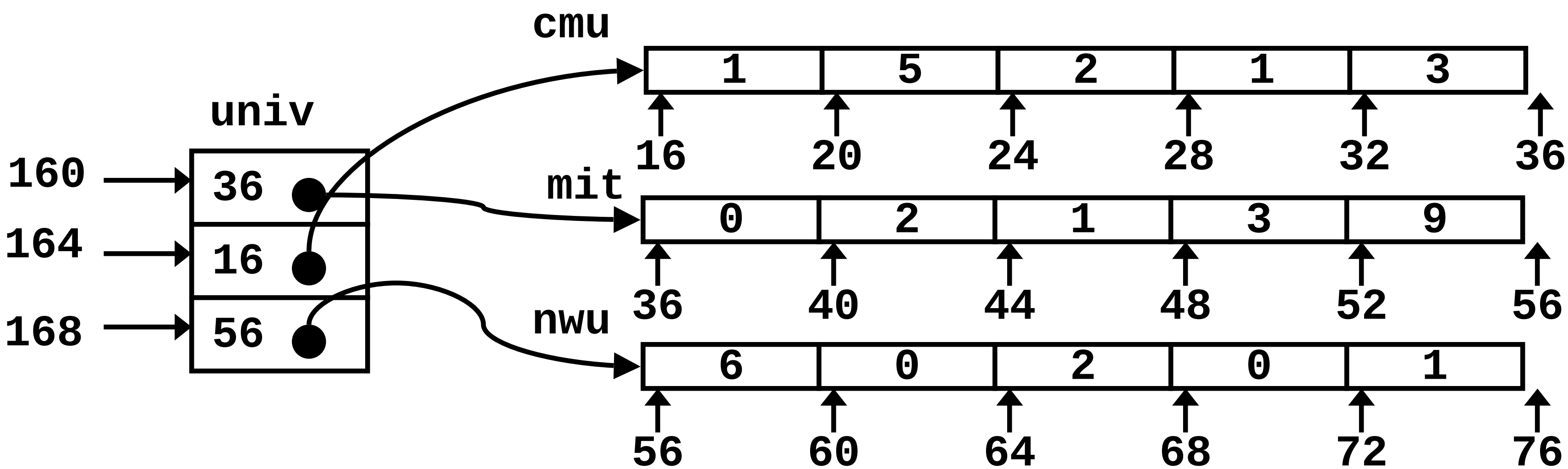
Computation

- **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

```
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]
```

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

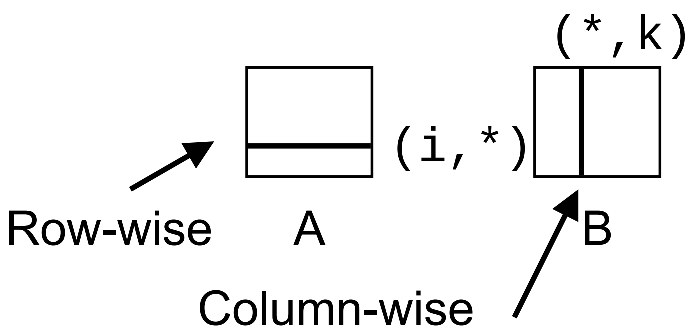
Using Nested Arrays

Strengths

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

Limitation

- **Only works if have 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 arbitrary 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];
}
```

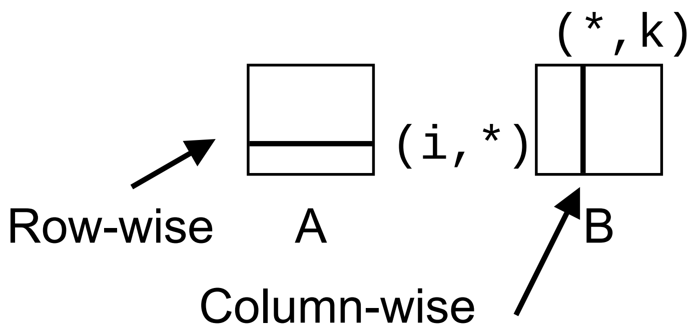
```
movl 12(%ebp),%eax       # i
movl 8(%ebp),%edx        # a
imull 20(%ebp),%eax      # n*i
addl 16(%ebp),%eax       # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```

Dynamic Array Multiplication

Without Optimizations

- **Multiplies**
  - 2 for subscripts
  - 1 for data
- **Adds**
  - 4 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`

Performance

- 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;
}
```

Dynamic Array Multiplication

```
{
  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;
}
```

%ecx	result
%edx	j
%esi	n
%ebx	jTnPk
Mem[-4(%ebp)]	iTn

```
.L44:                           # loop
  movl -4(%ebp),%eax            # iTn
  movl 8(%ebp),%edi             # a
  addl %edx,%eax                # iTn+j
  movl (%edi,%eax,4),%eax       # a[..]
  movl 12(%ebp),%edi            # b
  incl %edx                     # j++
  imull (%edi,%ebx,4),%eax      # b[..]*a[..]
  addl %eax,%ecx                # result += ..
  addl %esi,%ebx                # jTnPk += j
  cmpl %esi,%edx                # j : n
  jl .L44                       # if < goto loop
```

Inner Loop

Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code
  `zd2int`
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops
  - code motion
  - reduction in strength