Stack Discipline Examples, Arrays in Assembly

Topics

- Recursive factorial examples
  - Creating pointers to local variables
- Arrays

Announcements

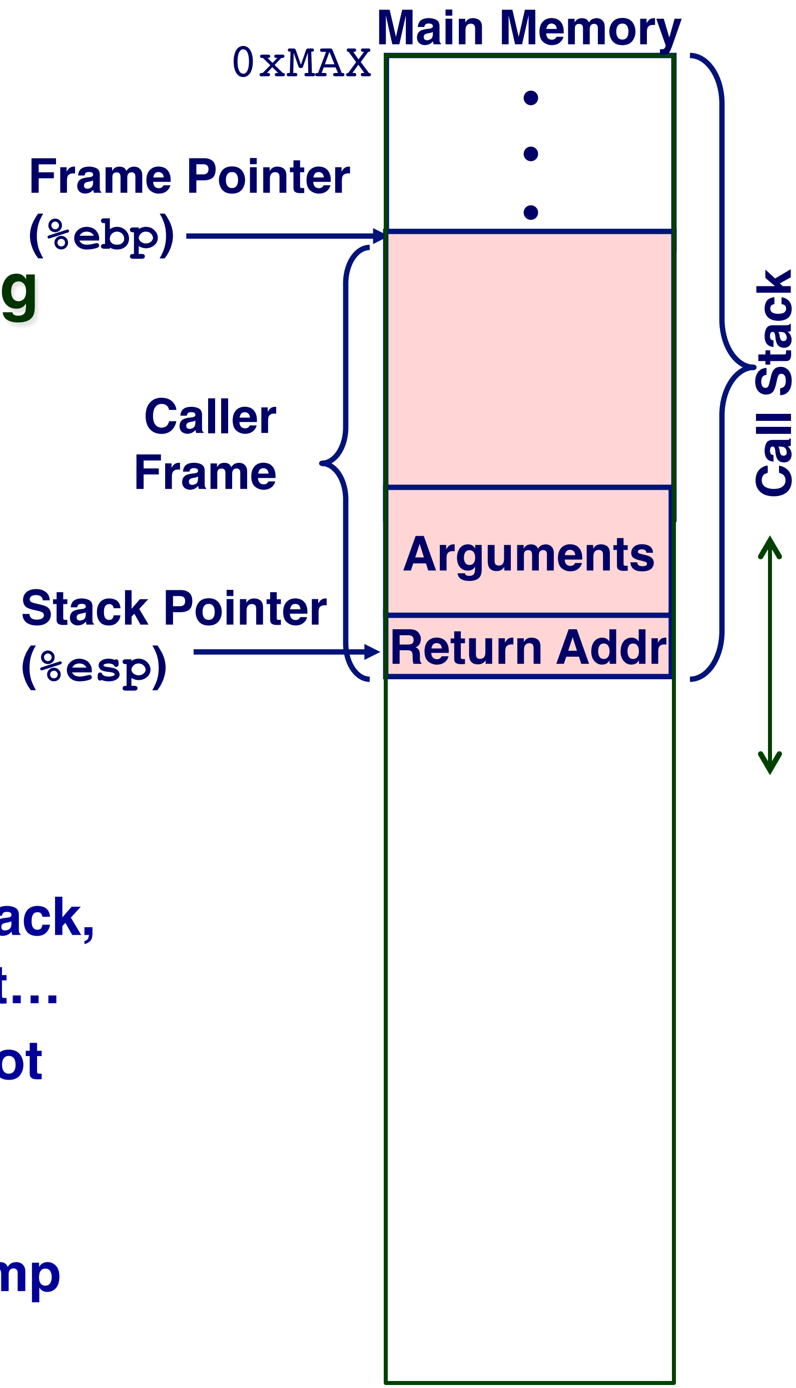
- **Bomb Lab is due Friday Oct 3 by 11:55 pm**
  - Reminder of TA office hours:
    - Thursdays 10:30-12:30 pm (Yogesh in CSEL)
    - Thursdays 3-5 pm (Pate in ~~ECCS 123~~ CSEL)
    - Fridays 10-12 noon (Abhishek in CSEL)
  - Extra credit secret bomb: add 7 points to your final lab grade
- **First midterm is probably Tuesday Oct 7**
- **Recitation Exercises #2 due Monday Sept 29**
- **Question on static variables**
  - Allocated like global variables in .data/.bss - See Chapter 7
- **Essential that you read the textbook in detail & do the practice problems**
  - Read Chapter 3.1-3.14, skip 3.12 for now

Summarizing Stack Discipline

Motivation for stacks – supporting function calls efficiently

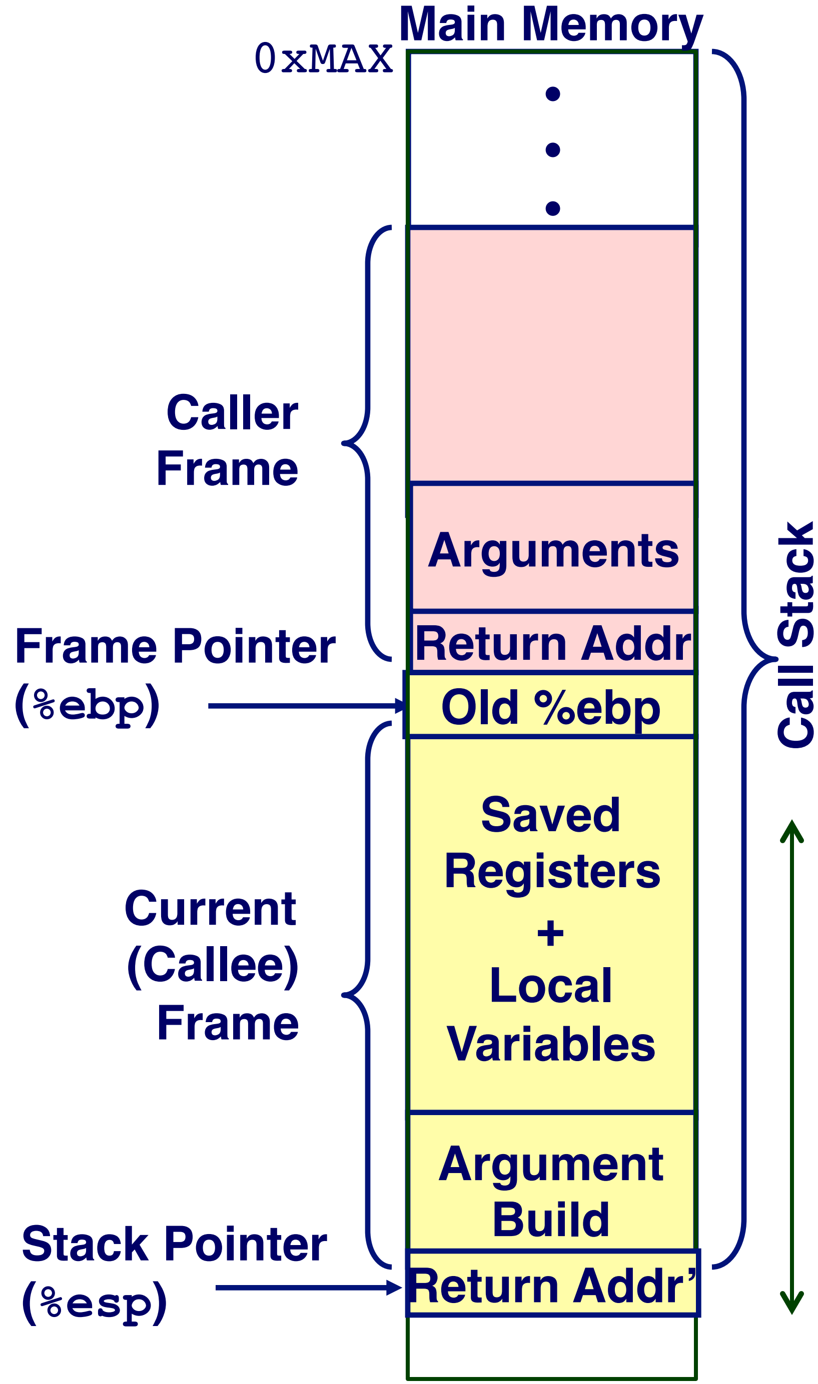
How stacks work:

- Stack pointer stored in register `%esp`, frame pointer in `%ebp`
- Manipulating stacks: `pushl`, `popl`, `call`, `ret`
- Before a function `call`:
  1. Push parameters onto the stack, last argument is pushed first…
  2. Save caller-save registers (not shown)
  3. Then `call` will push return address on the stack and jump into function



Summarizing Stack Discipline

- Inside the function call:
  1. Save frame pointer %ebp and slide %ebp down
  2. Save callee-save registers
  3. Allocate space for local variables
  4. On a return, restore callee-save registers, stack pointer, frame pointer, pop return address & jump back
- After the function call:
  - restore caller-save registers
  - continue execution…



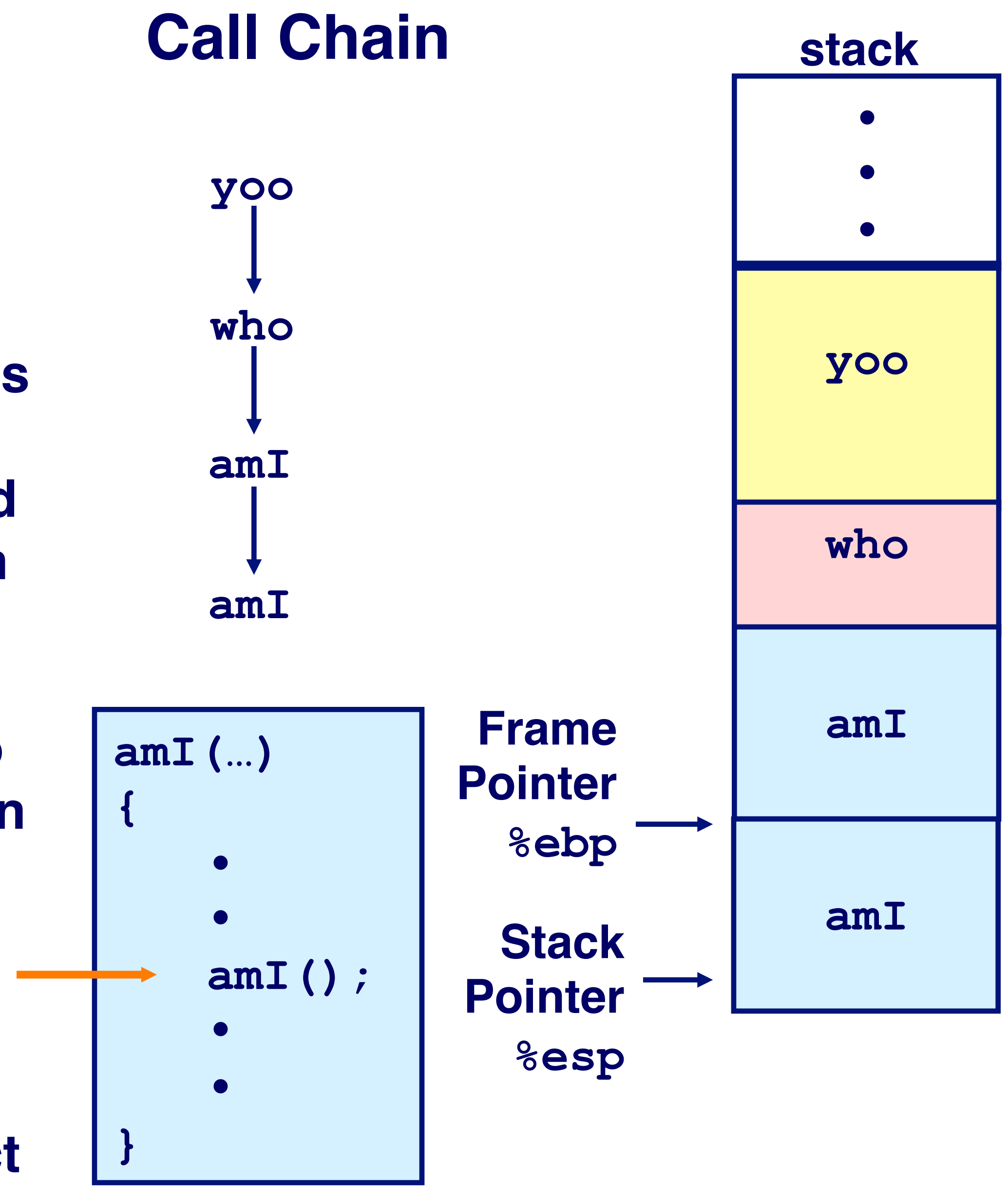
Recap…

Frame-based view of stack:

- Each function call pushes a frame onto the stack, and the frame is removed upon exiting the function

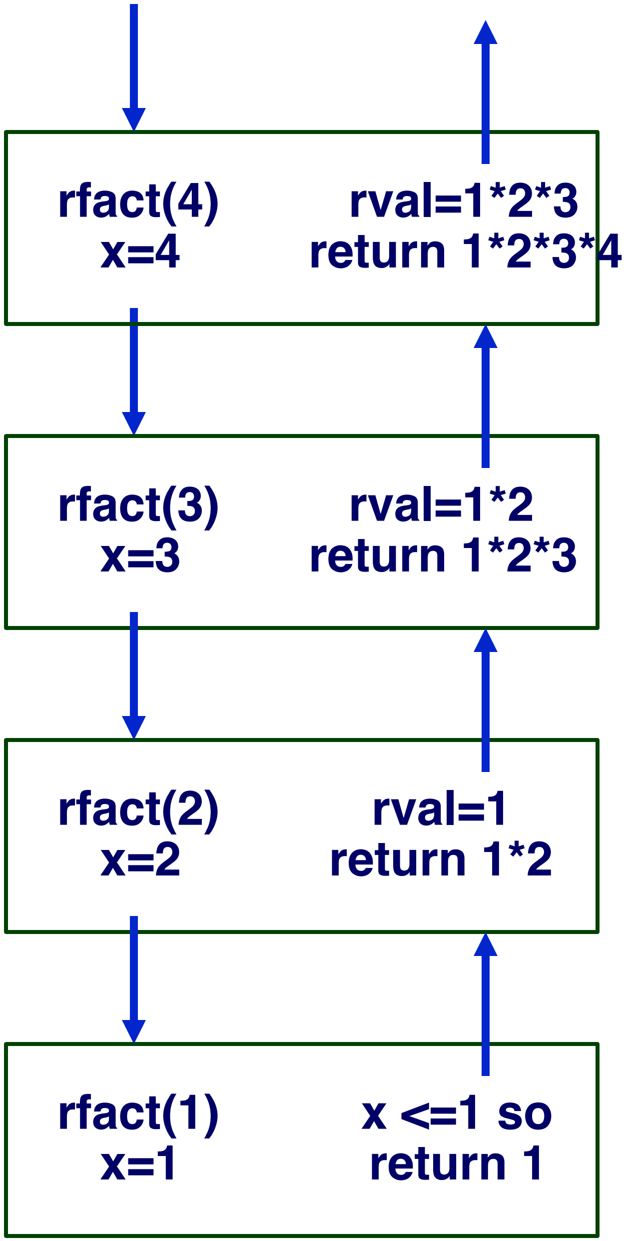
Recursion and stacks

- Each call of a function to itself puts a new frame on the stack
- This new frame is a new instance of the function
- Calculations affecting local variables only affect the frame that they're in



Recursive Factorial

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



Recursive Factorial

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

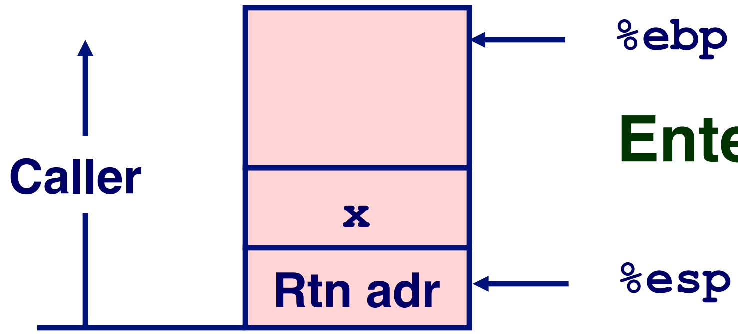
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    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
```

Registers

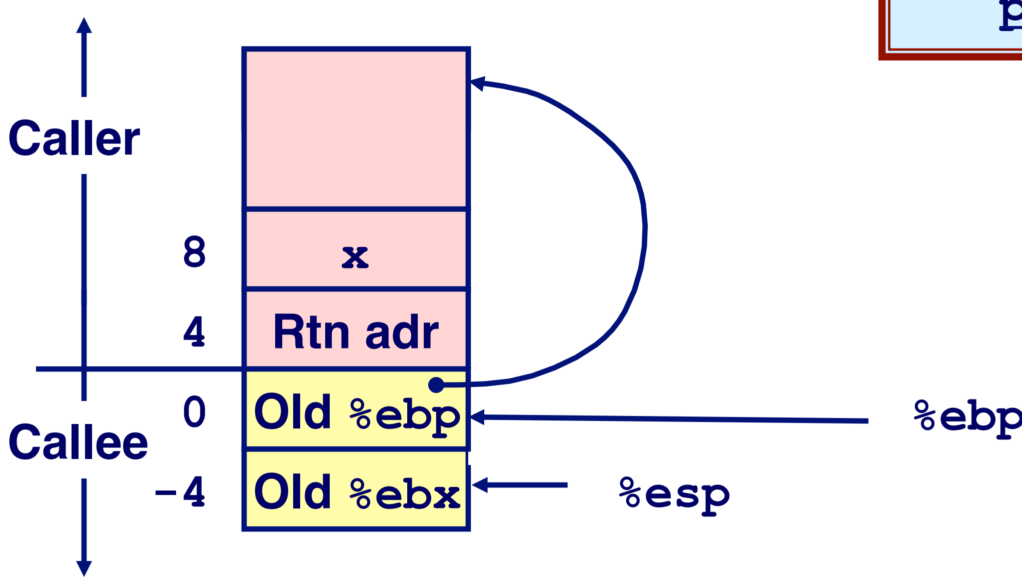
- `%ebx` used, but saved at beginning & restored at end
- `%eax` used without first saving, and stores the return value

Rfact Stack Setup



Entering Stack

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



Rfact Body

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

Recursion

```
  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:
```

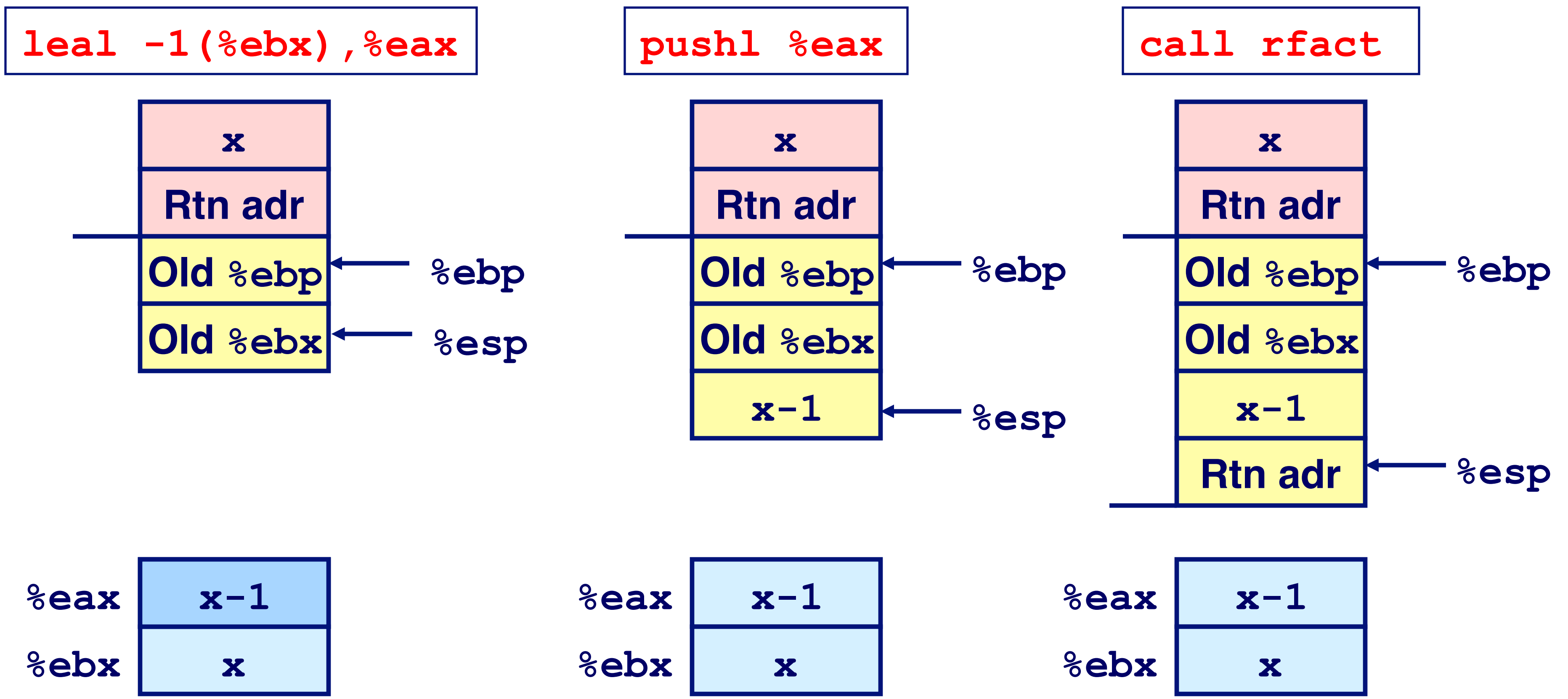
Registers

`%ebx` Stored value of x

`%eax`

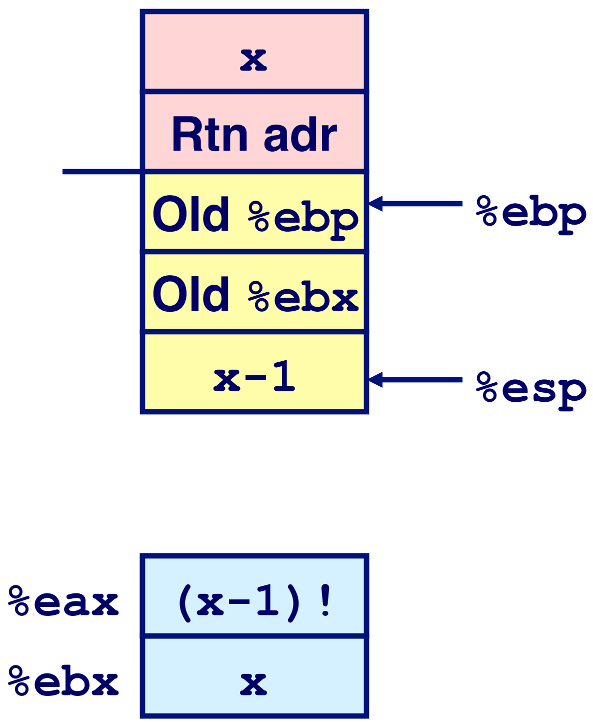
- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

Rfact Recursion

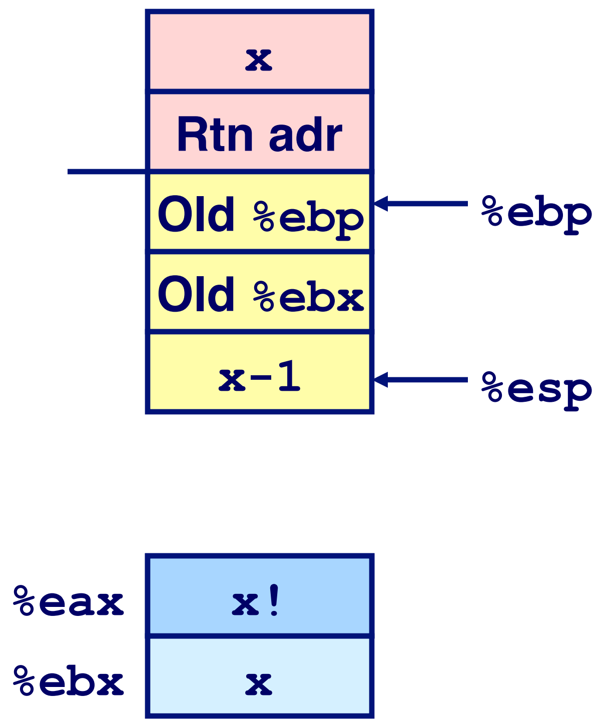


Rfact Result

Return from Call



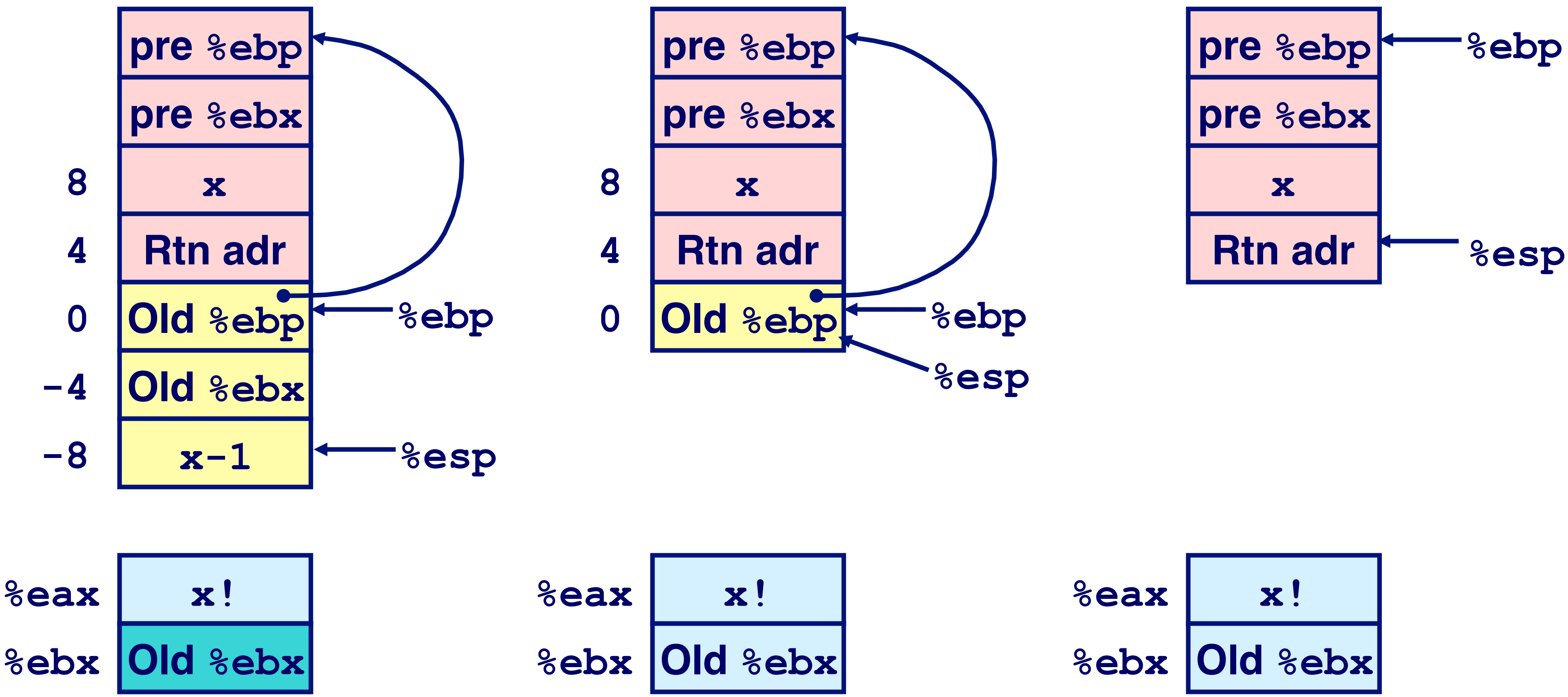
`imull %ebx,%eax`



Convince yourself that `rfact(x-1)` returns `(x-1)!` in register `%eax`

Rfact Completion

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



Recursion with Pointers

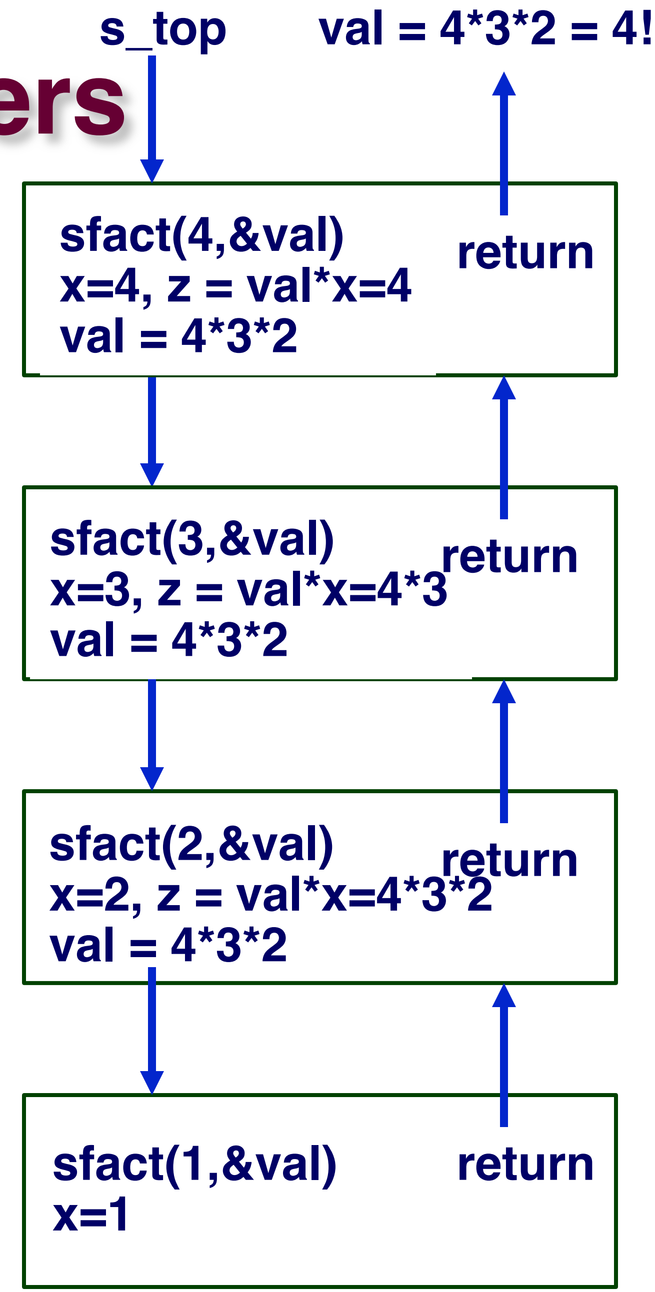
Top-Level Call

Note pointer

```
int s_top(int x)
{
  int val = 1;
  sfact(x, &val);
  return val;
}
```

Recursive Procedure

```
void sfact
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    sfact(x-1,accum);
  }
}
```



Note: this slide only makes sense with animation

Recursion: Pointer Creation

Top-Level Call

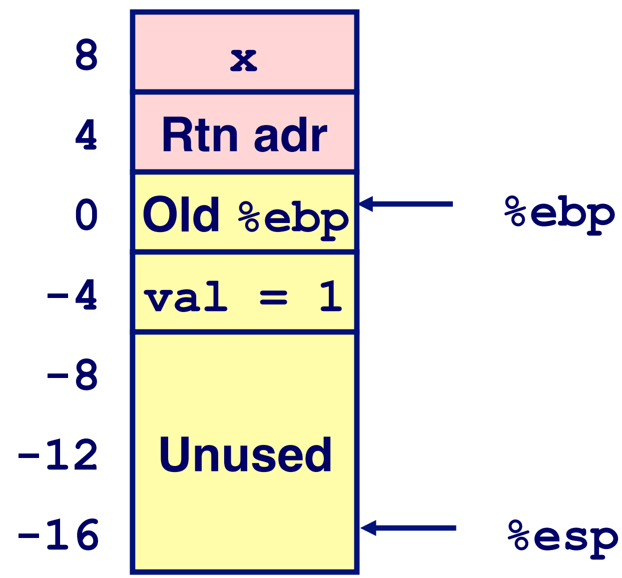
```
int s_top(int x)
{
  int val = 1;
  sfact(x, &val);
  return val;
}
```

Initial part of `s_top`

```
_s_top:
  pushl %ebp            # Save %ebp
  movl %esp,%ebp        # Set %ebp
  subl $16,%esp         # Add 16 bytes
  movl 8(%ebp),%edx     # edx = x
  movl $1,-4(%ebp)      # val = 1
```

Using Stack for Local Variable

- Local variable `val` is created and stored on stack
- Its address is passed into other procedures…
- … which enables those procedure to change the value of `val` (factorial product)



Recursion: Pointer Passing

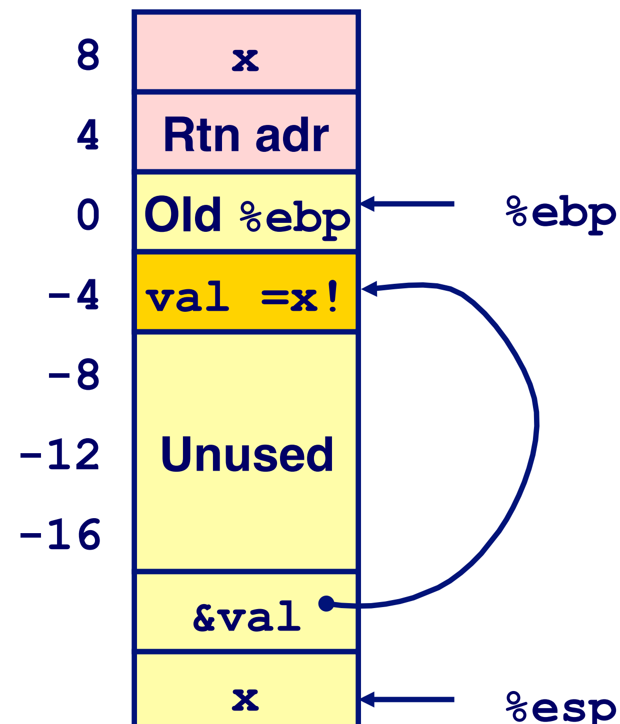
Top-Level Call

```
int s_top(int x)
{
  int val = 1;
  sfact(x, &val);
  return val;
}
```

Calling `sfact` from `s_top`

```
leal -4(%ebp),%eax # Compute &val
pushl %eax         # Push on stack
pushl %edx         # Push x
call sfact         # call
movl -4(%ebp),%eax # Return val
• • •              # Finish
```

Stack at time of call

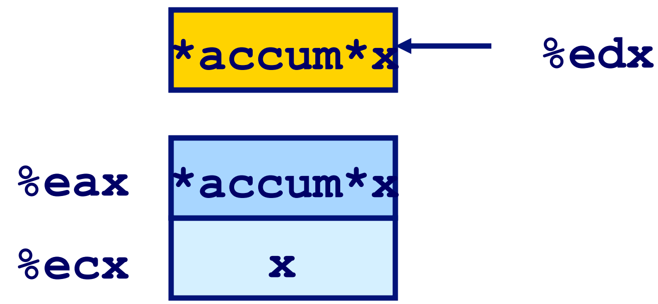


Before calling sfact():

- Create the pointer to `val`
  - = `-4(%ebp)`
- Push on stack as second argument

Recursion: Pointer Use

```
void sfact
   (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```




```
  • • •
  movl %ecx,%eax     # z = x
  imull (%edx),%eax  # z *= *accum
  movl %eax,(%edx)   # *accum = z
  • • •
```

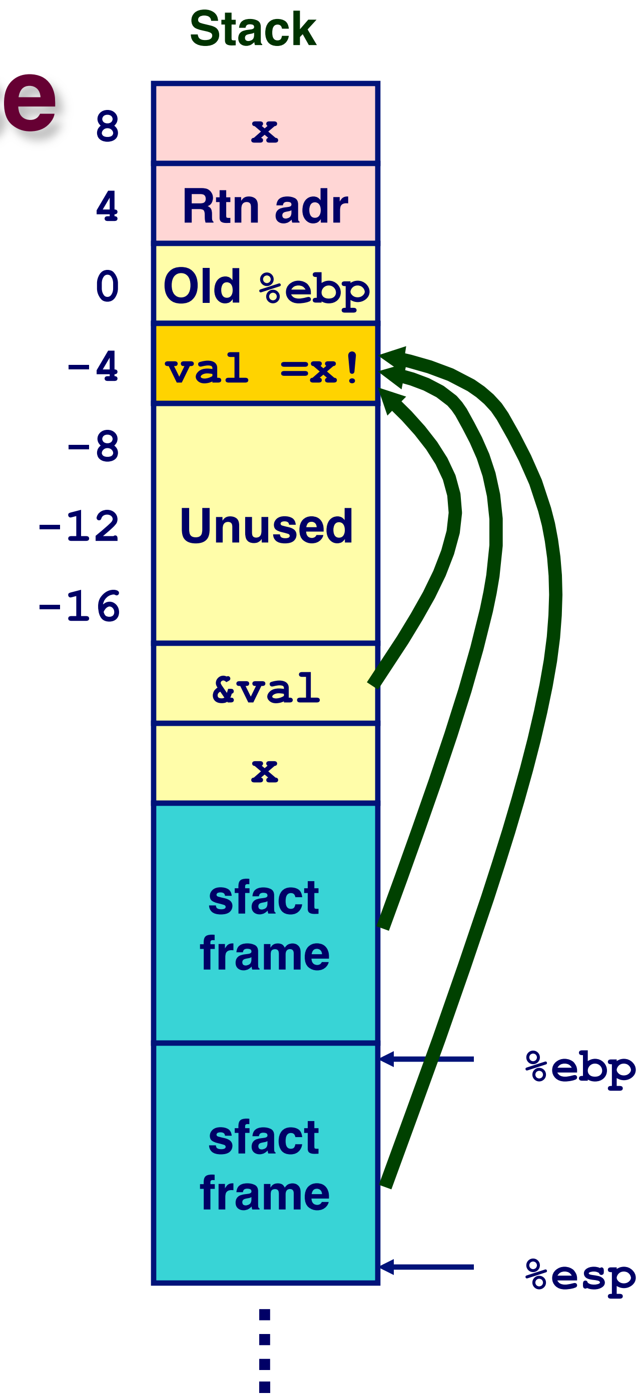
- Register `%ecx` holds `x`
- Register `%edx` holds pointer to `accum`
  - Use access `(%edx)` to reference memory

Recursion: Pointer Use

```
void sfact
   (int x, int *accum)
{
   • • •
     int z = *accum * x;
     *accum = z;
   • • •
}
```

```
   • • •
   movl %ecx,%eax     # z = x
   imull (%edx),%eax  # z *= *accum
   movl %eax,(%edx)   # *accum = z
   • • •
```

- Each recursive call computes a new z value in a new frame, and updates the partial product stored in local variable `val`, located in a different stack frame
- So pointers can be to temporary stack variables

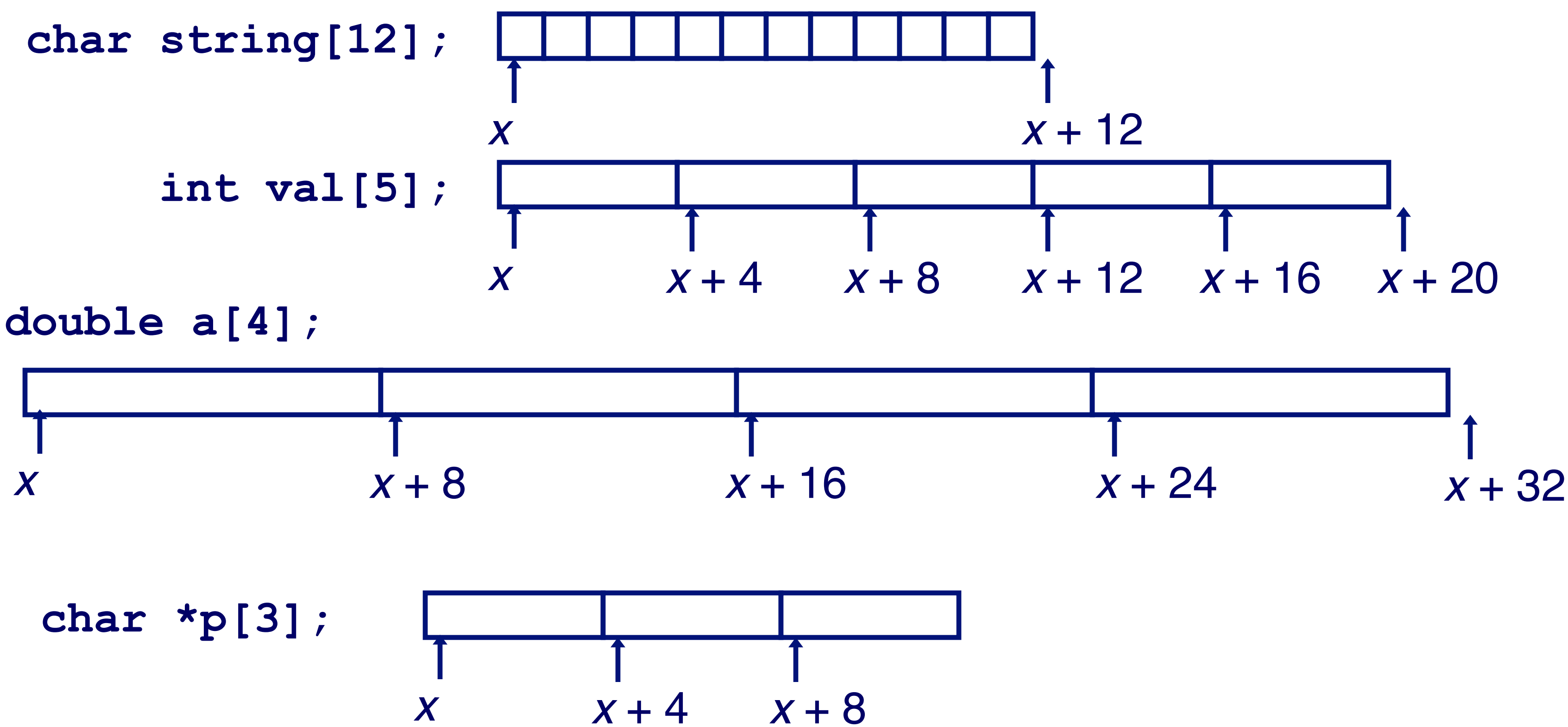


Array Allocation

Basic Principle

Type_T `Name_of_array[`*L*`];`

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



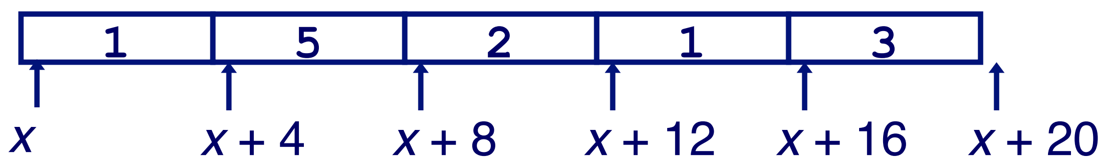
Array Access

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

`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*

Memory address of *i*' th index element of array

Array Access (2)

To access the *i*' th index element of an array A of type T:

- A[*i*] = *(A + sizeof(T)**i*)

If A is an array of ints, and address of A is stored in register %edx, and *i* is stored in register %ecx, then

- movl (%edx,%ecx,4), %eax

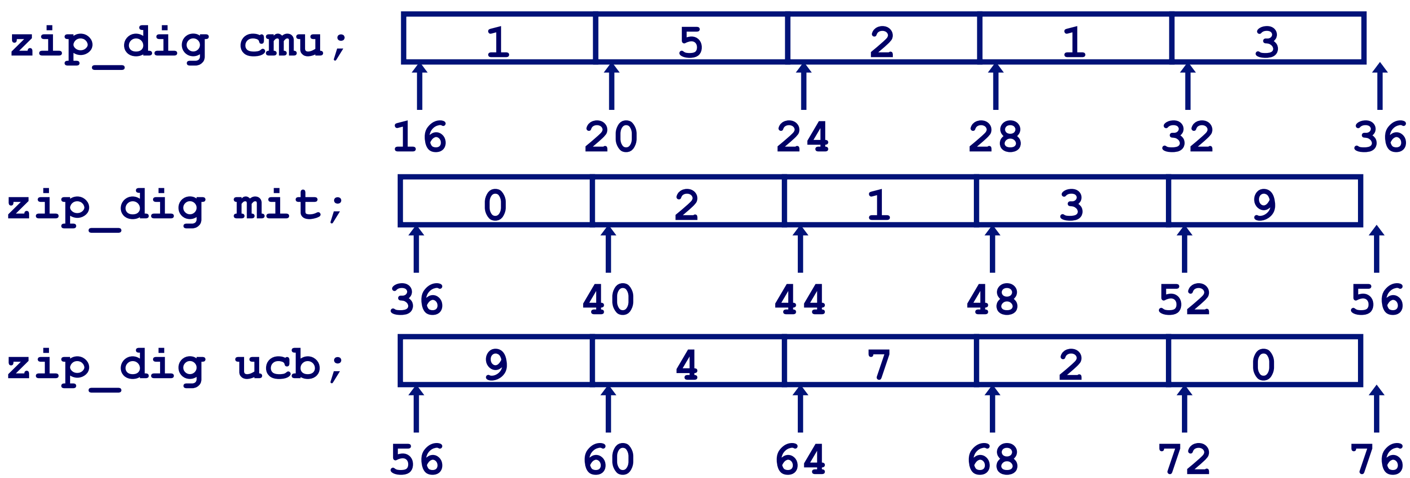
will compute pointer %edx+4*%ecx	// (A + sizeof(T)**i*)
And pull from memory (%edx+4*%ecx)	// *(A + sizeof(T)**i*)
Placing it into %eax	// = A[*i*]

So array access naturally fits with and uses the complex addressing mode provided by the CPU

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 };
```



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

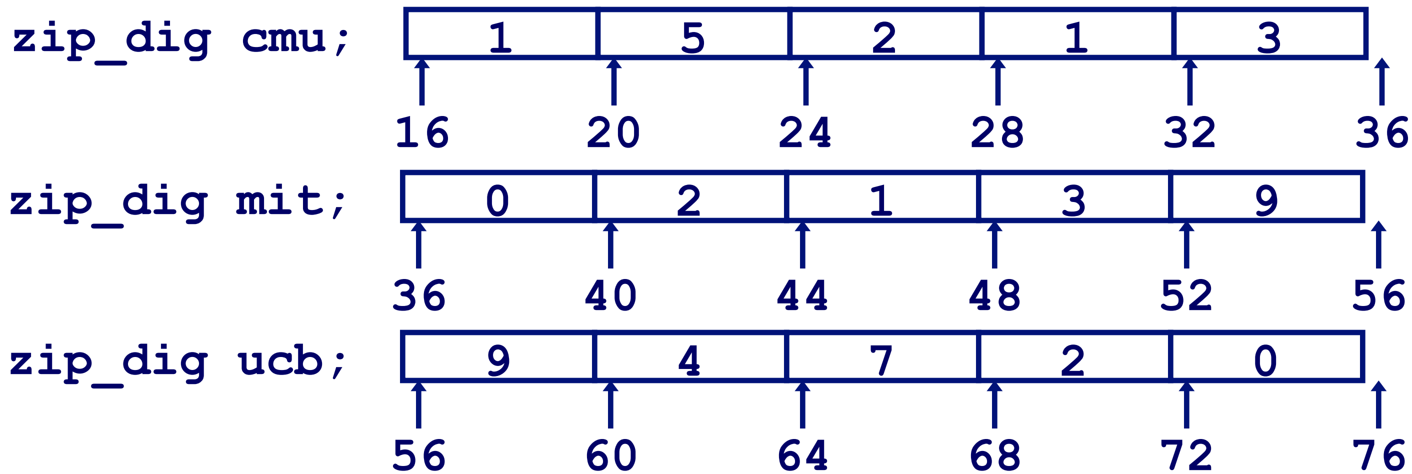
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

```
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 guaranteed relative allocation of different arrays**

Supplementary Slides

Summary

The Stack Makes Recursion Work

- Private storage for each *instance* of procedure call
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - `%ebp` and `%esp`
- Stack frame organization conventions

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`

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints