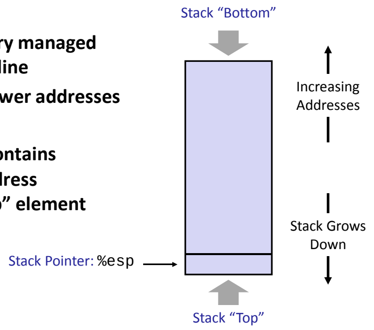
Computer Systems I

Class 7

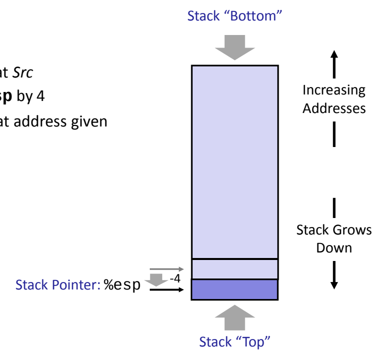
IA32 Stack

- **Region of memory managed with stack discipline**
- **Grows toward lower addresses**
- **Register `%esp` contains lowest stack address = address of "top" element**



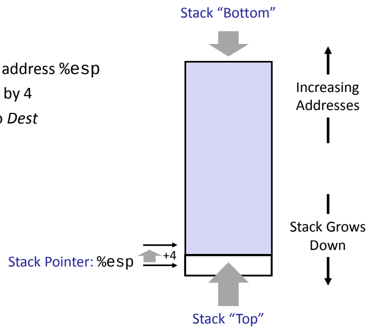
IA32 Stack: Push

- **`pushl` *Src***
  - Fetch operand at *Src*
  - Decrement **`%esp`** by 4
  - Write operand at address given by **`%esp`**



IA32 Stack: Pop

- `popl` *Dest*
  - Read operand at address `%esp`
  - Increment `%esp` by 4
  - Write operand to *Dest*



Systems I

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call:** `call` ***`label`***
  - Push return address on stack
  - Jump to ***`label`***
- **Return address:**
  - Address of instruction beyond `call`
  - Example from disassembly

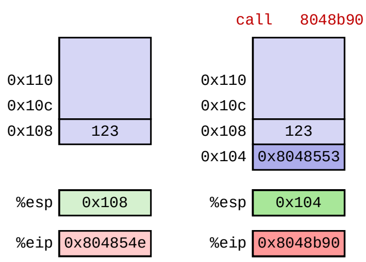
```
804854e:   e8 3d 06 00 00     call   8048b90 <main>
8048553:   50                 pushl  %eax
```

  - Return address = **`0x8048553`**
- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address

Systems I

Procedure Call Example

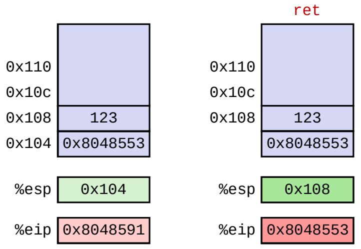
```
804854e:    e8 3d 06 00 00       call   8048b90 <main>
8048553:    50                   pushl  %eax
```



`%eip`: program counter

Procedure Return Example

```
8048591:    c3                    ret
```

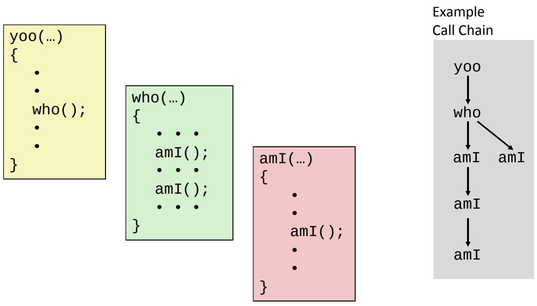


`%eip`: *program counter*

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- **Stack allocated in *Frames***
  - state for single procedure instantiation

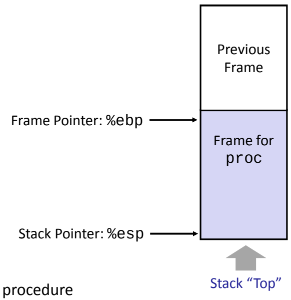
Call Chain Example



Procedure `amI` is recursive

Stack Frames

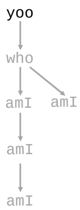
- **Contents**
  - Local variables
  - Return information
  - Temporary space

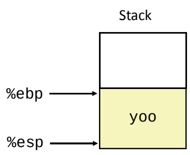


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

Example

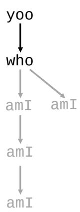
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

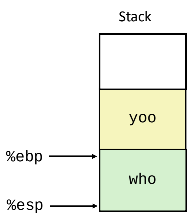




Example

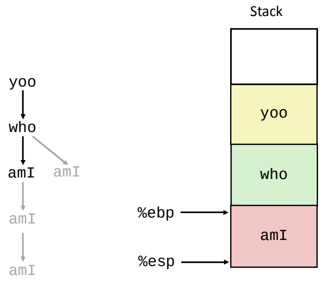
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```





Example

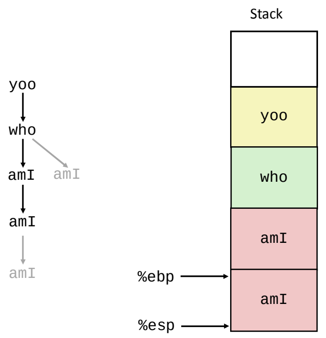
```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```



Systems I

Example

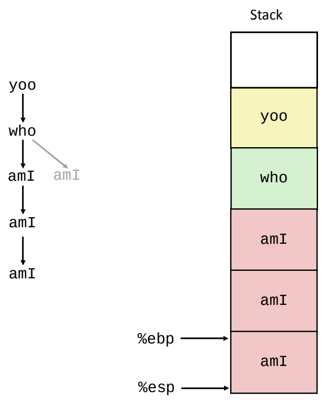
```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

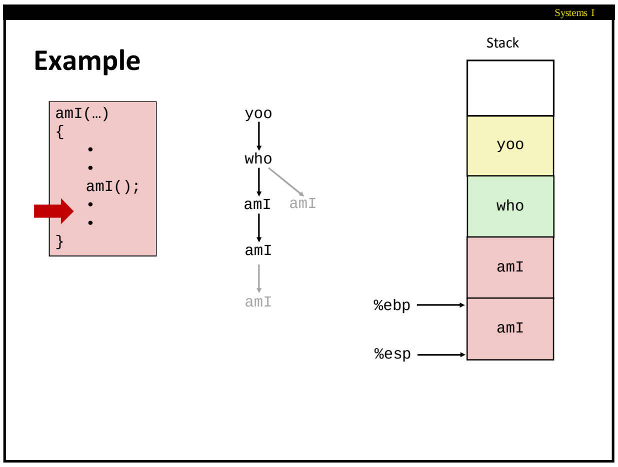


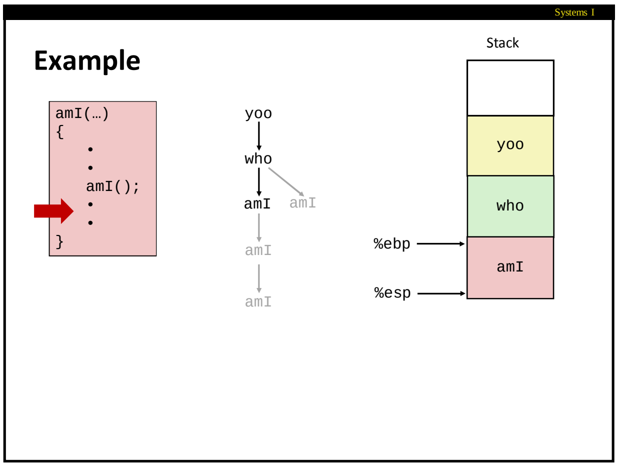
Systems I

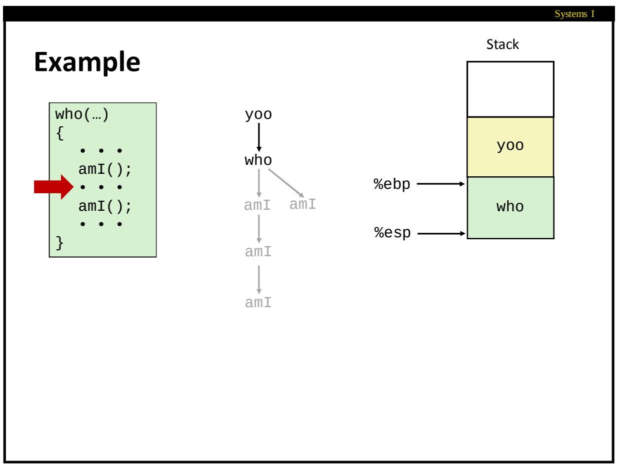
Example

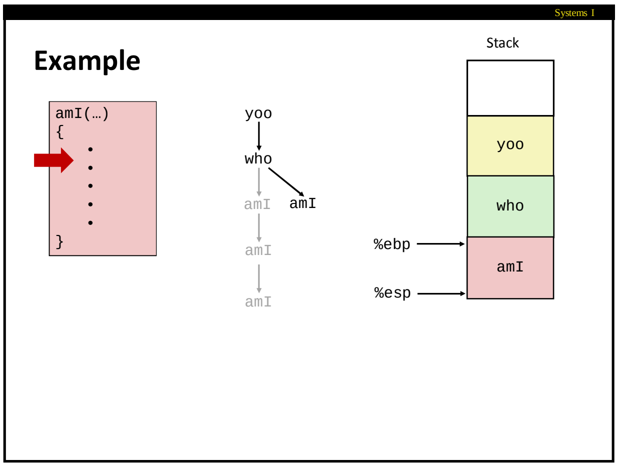
```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

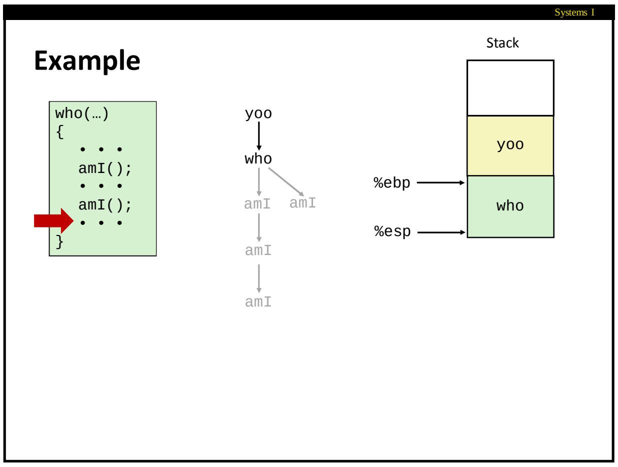


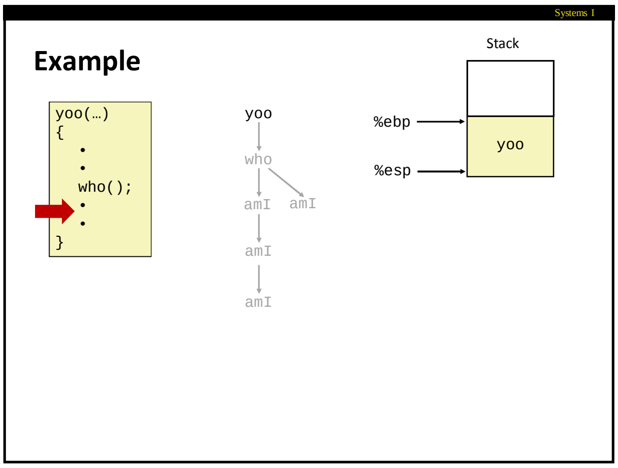

Systems I
Example
Stack
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
%ebp
%esp


Systems I
Example
Stack
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%ebp
%esp


Systems I
Example
Stack
who(…)
{
• • •
amI();
• • •
amI();
• • •
}
yoo
who
amI
amI
amI
amI
yoo
who
%ebp
%esp

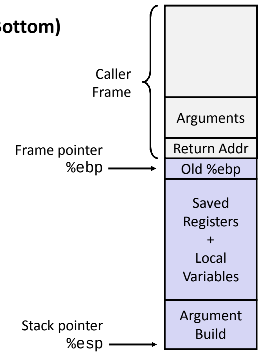

Systems I
Example
amI(…)
{
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%ebp
%esp


Systems I
Example
who(…)
{
• • •
amI();
• • •
amI();
• • •
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
%ebp
%esp


Systems I
Example
yoo(…)
{
who();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
%ebp
%esp

IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer
- **Caller Stack Frame**
  - Return address
  - Pushed by `call` instruction
  - Arguments for this call



Revisiting swap

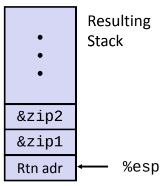
```
int zip1 = 15213;
int zip2 = 91125;

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

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

Calling swap from `call_swap`

```
call_swap:
    • • •
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    • • •
```



Revisiting swap (current gcc)

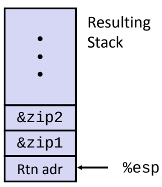
```
int zip1 = 15213;
int zip2 = 91125;

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

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

Calling swap from `call_swap`

```
  . . .
  subl    $24, %esp
  movl    $zip2, 4(%esp)
  movl    $zip1, (%esp)
  call    swap
  . . .
```



Revisiting swap

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

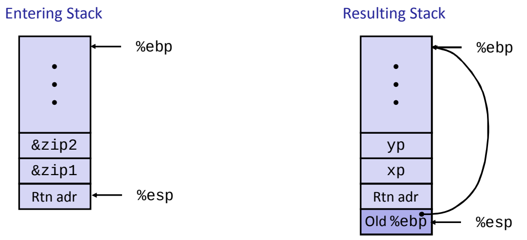
```
swap:
   pushl %ebp          } Set
   movl %esp,%ebp      } Up
   pushl %ebx

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

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

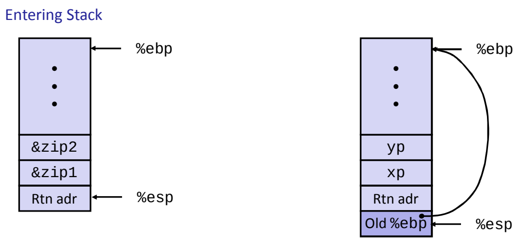
Do on blackboard?

swap Setup #1



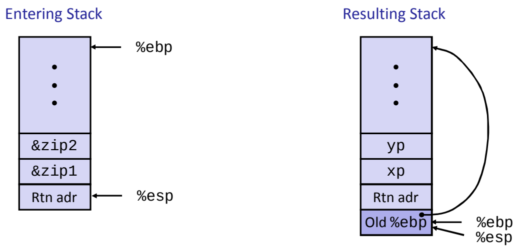

```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
```


swap Setup #1



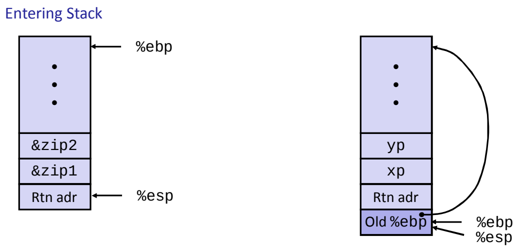

```
swap:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
```

swap Setup #1



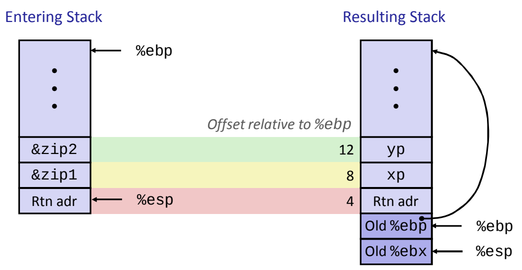

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```


swap Setup #1




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


swap Setup #1

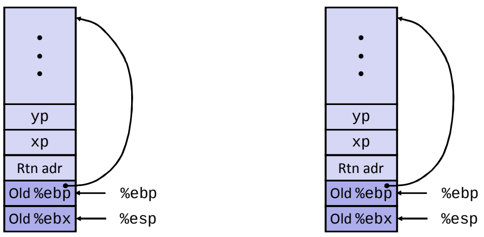



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

swap Finish #1

swap's Stack

Resulting Stack

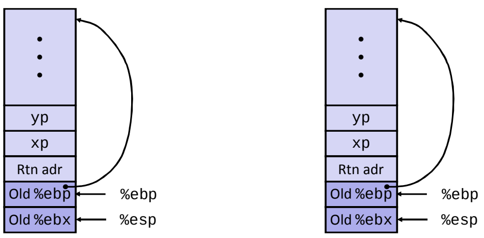



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

swap Finish #2

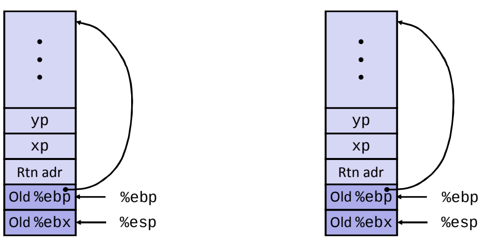
swap's Stack




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


swap Finish #2

swap's Stack

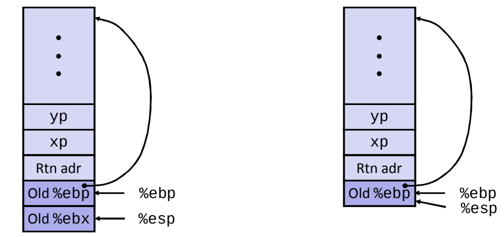



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #2

swap's Stack

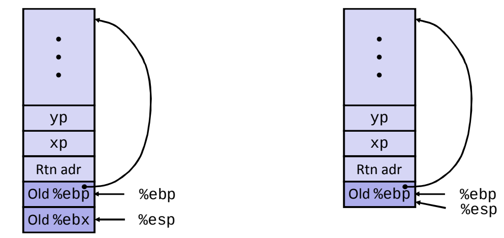
Resulting Stack




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


swap Finish #2

swap's Stack

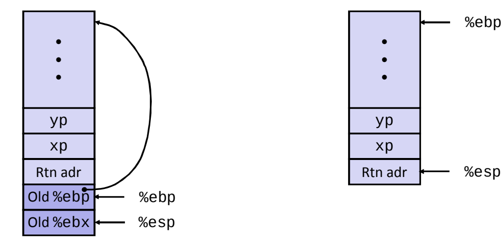



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```


swap Finish #3

swap's Stack

Resulting Stack

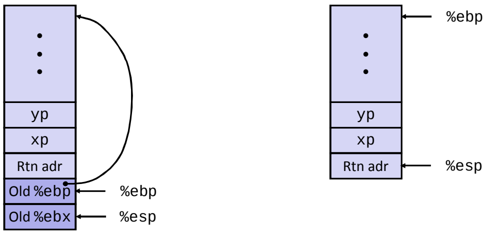



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

swap's Stack

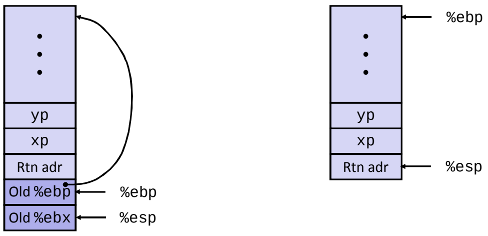
Resulting Stack




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


swap Finish #4

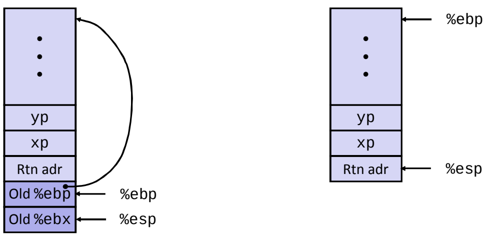
swap's Stack




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

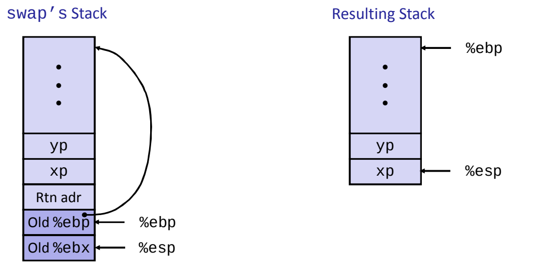

swap Finish #4

swap's Stack




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

swap Finish #4




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

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

Disassembled swap

```
080483a4 <swap>:
 80483a4:   55            push   %ebp
 80483a5:   89 e5         mov    %esp,%ebp
 80483a7:   53            push   %ebx
 80483a8:   8b 55 08      mov    0x8(%ebp),%edx
 80483ab:   8b 4d 0c      mov    0xc(%ebp),%ecx
 80483ae:   8b 1a         mov    (%edx),%ebx
 80483b0:   8b 01         mov    (%ecx),%eax
 80483b2:   89 02         mov    %eax,(%edx)
 80483b4:   89 19         mov    %ebx,(%ecx)
 80483b6:   5b            pop    %ebx
 80483b7:   c9            leave
 80483b8:   c3            ret
```

Calling Code

```
 8048409:   e8 96 ff ff ff    call 80483a4 <swap>
 804840e:   8b 45 f8          mov  0xfffffff8(%ebp),%eax
```


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**

Register Saving Conventions

- **When procedure yoo calls who:**
  - **yoo** is the ***caller***
  - **who** is the ***callee***
- **Can register be used for temporary storage?**
- **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

- **`%eax, %edx, %ecx`**
  - Caller saves prior to call if values are used later
- **`%eax`**
  - also used to return integer value
- **`%ebx, %esi, %edi`**
  - Callee saves if wants to use them
- **`%esp, %ebp`**
  - special

Caller-Save Temporaries	%eax
	%edx
	%ecx
Callee-Save Temporaries	%ebx
	%esi
	%edi
Special	%esp
	%ebp

Recursive Factorial

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

- **Registers**
  - **`%eax`** used without first saving
  - **`%ebx`** used, but saved at beginning & restore at end

```
.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
```

Systems I

Pointer Code

Recursive Procedure

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

Top-Level Call

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

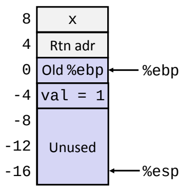
- **Pass pointer to update location**

Systems I

Creating & Initializing Pointer

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

- **Variable `val` must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as `-4(%ebp)`**
- **Push on stack as second argument**



Initial part of `sfact`

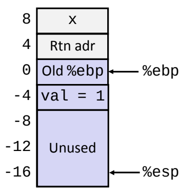
```
_sfact:
  pushl %ebp
  movl %esp,%ebp
  subl $16,%esp
  movl 8(%ebp),%edx
  movl $1,-4(%ebp)
```

Systems I

Creating & Initializing Pointer

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

- **Variable `val` must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as `-4(%ebp)`**
- **Push on stack as second argument**

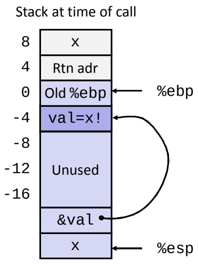


Initial part of `sfact`

```
_sfact:
  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
```

Passing Pointer

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

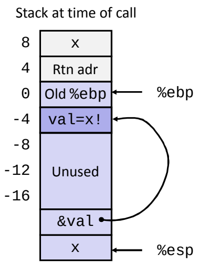


Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax
pushl %eax
pushl %edx
call s_helper
movl -4(%ebp),%eax
• • •
```


Passing Pointer

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

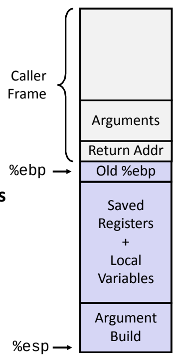


Calling `s_helper` from `sfact`

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


IA 32 Procedure 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
  - 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



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)

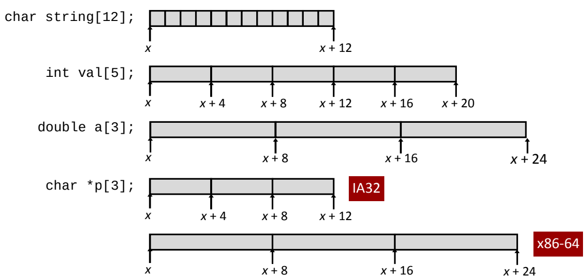
- **Floating Point**
  - Stored & operated on in floating point registers

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

Array Allocation

- **Basic Principle**

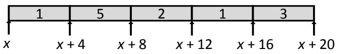
  T **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**(*T*) bytes



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];`



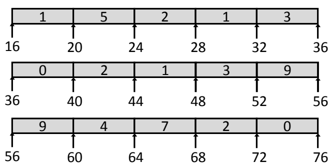
- **Reference** **Type** **Value**
  val[4]
  val
  val+1
  &val[2]
  val[5]
  ***(val+1)**
  val + ***i***

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 };
```

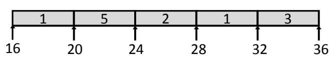
`zip_dig cmu;`
`zip_dig mit;`
`zip_dig ucb;`



- **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

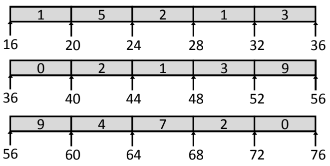
```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax   # z[dig]
```

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

Systems I

Referencing Examples

`zip_dig cmu;`
`zip_dig mit;`
`zip_dig ucb;`

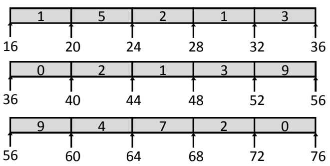


Reference	Address	Value	Guaranteed?
mit[3]			
mit[5]			
mit[-1]			
cmu[15]			

Systems I

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

Systems I

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;
}
```

Systems I

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
```

Systems I

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
```

Systems I

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

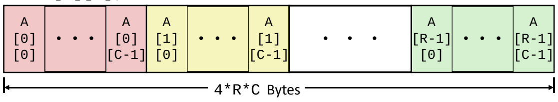
- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

Multidimensional (Nested) Arrays

- **Declaration**
  - *T* **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

```
A[0][0]    • • •    A[0][C-1]
   •                    •
   •                    •
   •                    •
A[R-1][0]  • • •  A[R-1][C-1]
```

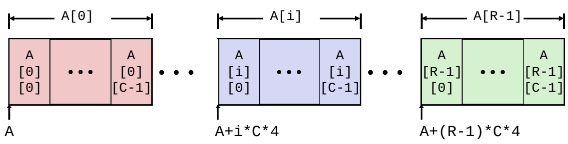
`int A[R][C];`



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)`**