An artificially intelligent virtual gamer created by computer scientists at The University of Texas at Austin has won the BotPrize by convincing a panel of judges that it was more human-like than half the humans it competed against.

The bots face off in a tournament against one another and about an equal number of humans, with each player trying to score points by eliminating its opponents. Each player also has a "judging gun" in addition to its usual complement of weapons. That gun is used to tag opponents as human or bot.

The winning bots both achieved a humanness rating of 52 percent. Human players received an average humanness rating of only 40 percent.

http://www.sciencedaily.com/releases/2012/09/120926133235.htm

Conditional Branch Example

```
int absdiff(int x, int y)
{
  if (x <= y)
    return y-x;
  return x-y;
}
```

```
int absdiff2(int x, int y)
{
  if (x <= y)
    return y-x;
  else
    return x-y;
}
```

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- ***Body:***

  {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }

- ***Test*** returns integer

  = 0 interpreted as false

  ≠0 interpreted as true

“Do-While” Loop Compilation

```
fact_goto:
  pushl %ebp            # Setup
  movl %esp,%ebp        # Setup
  movl $1,%eax
  movl 8(%ebp),%edx

.L11:
  imull %edx,%eax
  decl %edx
  cmpl $1,%edx
  jg .L11

  movl %ebp,%esp        # Finish
  popl %ebp             # Finish
  ret                   # Finish
```

Registers:

%edx	**?**
%eax	**?**

"Do-While" Loop Compilation

Registers:

%edx	**x**
%eax	**result**

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp              # Setup
  movl %esp,%ebp          # Setup
  movl $1,%eax            # eax = 1
  movl 8(%ebp),%edx       # edx = x

.L11:
  imull %edx,%eax         # result *= x
  decl %edx               # x--
  cmpl $1,%edx            # Compare x : 1
  jg .L11                 # if > goto loop

  movl %ebp,%esp          # Finish
  popl %ebp               # Finish
  ret                     # Finish
```

“Do-While” Loop Example

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

“While” Loop Example

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version #1

```
int fact_while_goto(int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Alternative "While" Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version #2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

```
1   int loop_while(int a, int b)
2   {
3       int result = 1;
4       while (a < b) {
5           result *= (a+b);
6           a++;
7       }
8       return result;
9   }
```

a at %ebp+8, b at %ebp+12

```
1     movl    8(%ebp), %ecx
2     movl    12(%ebp), %ebx
3     movl    $1, %eax
4     cmpl    %ebx, %ecx
5     jge     .L11
6     leal    (%ebx,%ecx), %edx
7     movl    $1, %eax
8   .L12:
9     imull   %edx, %eax
10    addl    $1, %ecx
11    addl    $1, %edx
12    cmpl    %ecx, %ebx
13    jg      .L12
14  .L11:
```

1. %edx, on lines 6 and 11, seems like a new variable. How does it relate to the C code?
2. How is each register used?
3. Annotate the assembly
4. Write a goto version of the C function that mimics the assembly

```
Arguments: a at %ebp+8, b at %ebp+12
Registers: a in %ecx, b in %ebx, result in %eax, %edx set to apb (a+b)
    movl    8(%ebp), %ecx          Get a
    movl    12(%ebp), %ebx         Get b
    movl    $1, %eax               Set result = 1
    cmpl    %ebx, %ecx             Compare a:b
    jge     .L11                   If >=, goto done
    leal    (%ebx,%ecx), %edx      Compute apb = a+b
    movl    $1, %eax               Set result = 1
  .L12:                          loop:
    imull   %edx, %eax             Compute result *= apb
    addl    $1, %ecx               Compute a++
    addl    $1, %edx               Compute apb++
    cmpl    %ecx, %ebx             Compare b:a
    jg      .L12                   If >, goto loop
  .L11:                          done:
    Return result
```

```
int loop_while_goto(int a, int b)
{
    int result = 1;
    if (a >= b)
        goto done;
    /* apb has same value as a+b in original code */
    int apb = a+b;
 loop:
    result *= apb;
    a++;
    apb++;
    if (b > a)
        goto loop;
 done:
    return result;
}
```

General "While" Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```

Goto Version

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- `for` loops compiled in similar fashion

Goto (Previous) Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Jump-to-Middle Example

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

```
# x in %edx, result in %eax
  jmp    .L34        #   goto Middle
.L35:                # Loop:
  imull %edx, %eax #   result *= x
  decl   %edx        #   x--
.L34:                # Middle:
  cmpl   $1, %edx    #   x:1
  jg     .L35        #   if >, goto Loop
```

Implementing Loops

- IA32
  - All loops translated into form based on “do-while”
- x86-64
  - Also make use of “jump to middle”
- Why the difference
  - IA32 compiler developed for machine where all operations costly
  - x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead

“For” Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

"For" Loop → … → Goto

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while(Test);
done:
```

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

“For” Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

- Initial test can be optimized away

- http://blog.natekohl.net/abomination-and-astonishment/

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `-march=686` for IA32
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```

Conditional Move Example: x86-64

```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

x in %edi

y in %esi

```
absdiff:
  movl    %edi, %edx
  subl    %esi, %edx   # tval = x-y
  movl    %esi, %eax
  subl    %edi, %eax   # result = y-x
  cmpl    %esi, %edi   # Compare x:y
  cmovg   %edx, %eax   # If >, result = tval
  ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Procedure Control Flow

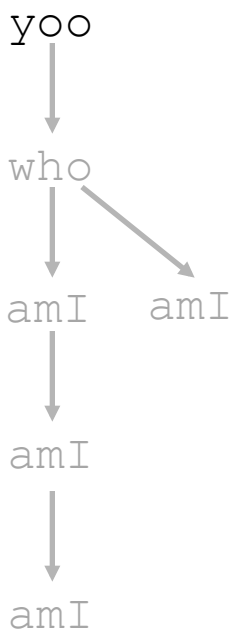
- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of instruction beyond **`call`**
  - Example from disassembly

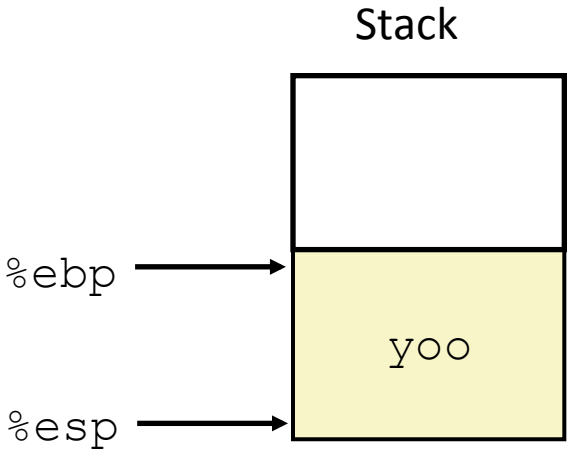
```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```

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

Example

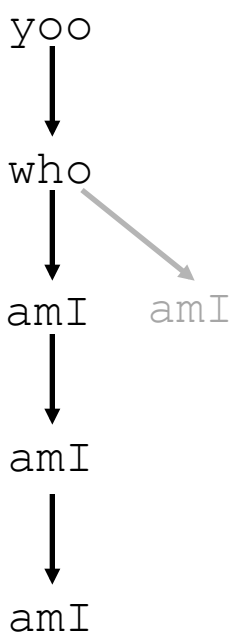
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

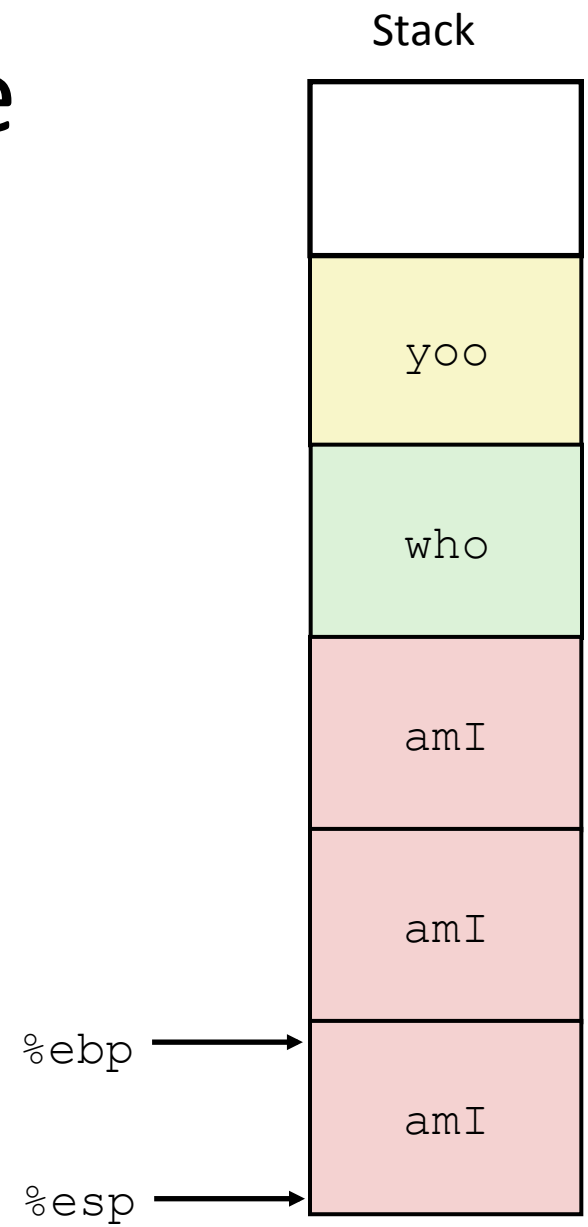




Example

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

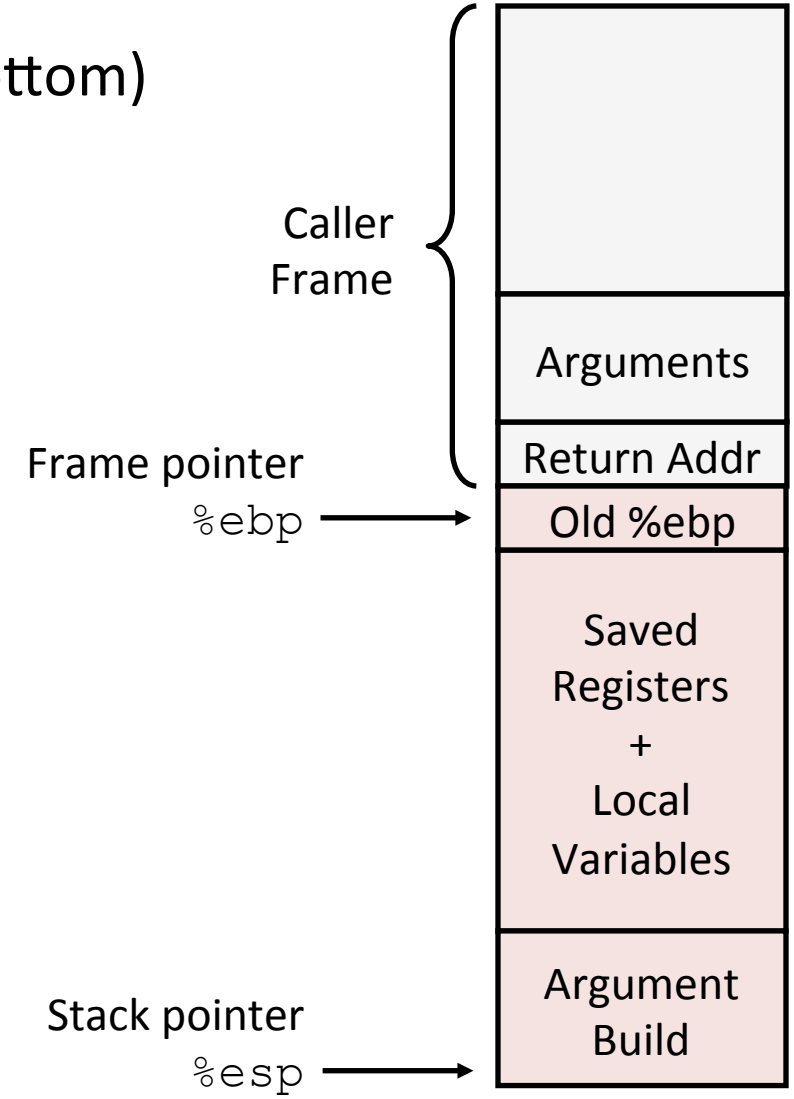




IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    - If can't keep in registers
    - If arrays/structures
    - If & applied to local variable
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
  - Pushed by **`call`** instruction
  - Arguments for this call



Revisiting swap

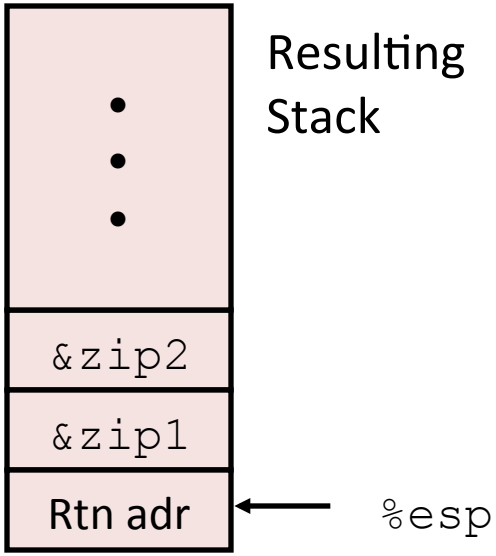
```
int zip1 = 15213;
int zip2 = 91125;

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

Calling swap from call_swap

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

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



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 %esp,%ebp
    popl %ebp
    ret
```

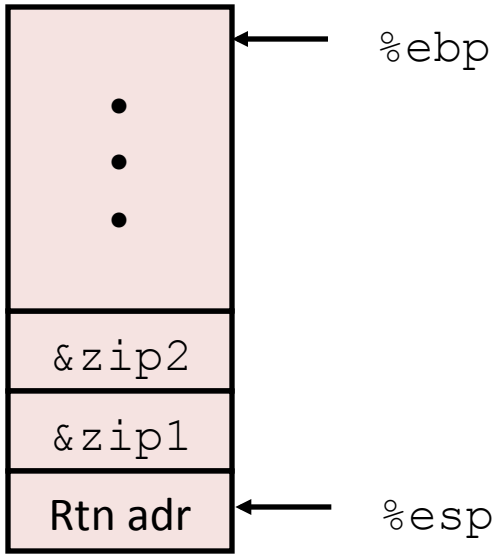
Set Up

Body

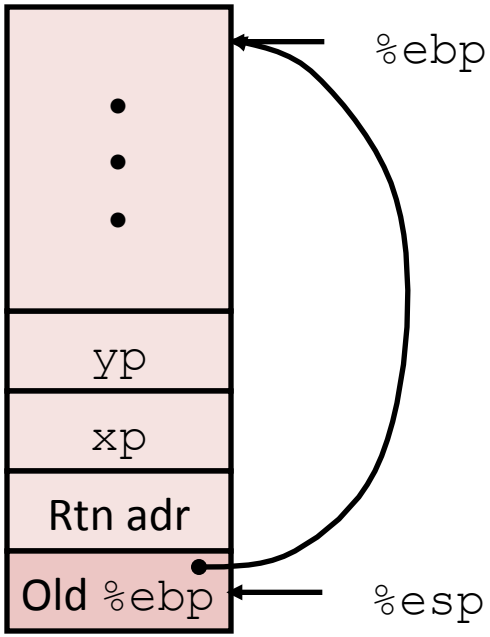
Finish

swap Setup #1

Entering Stack



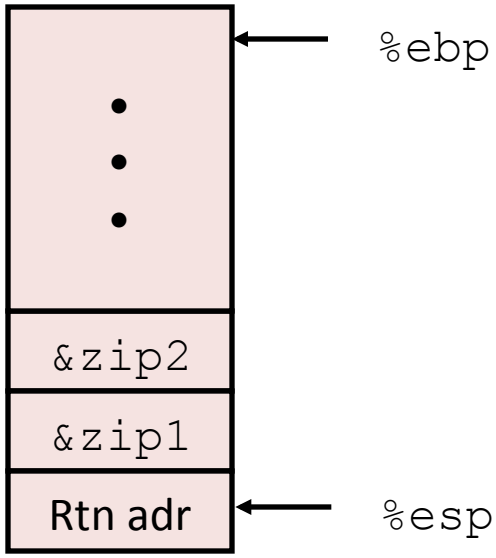
Resulting Stack

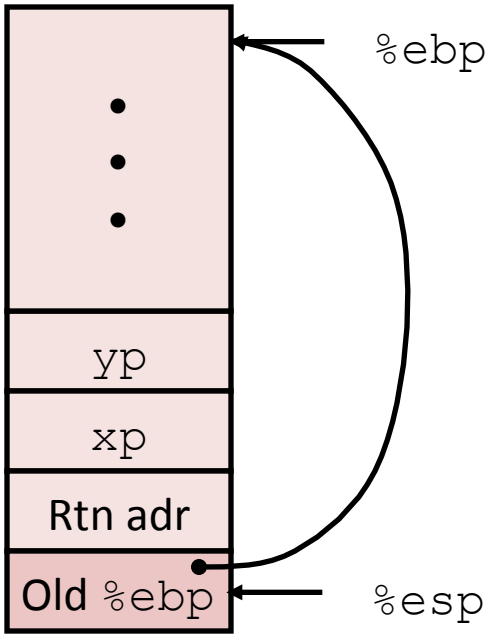



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack

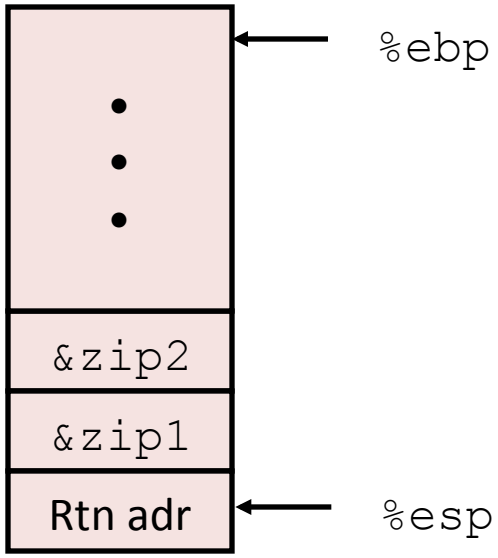




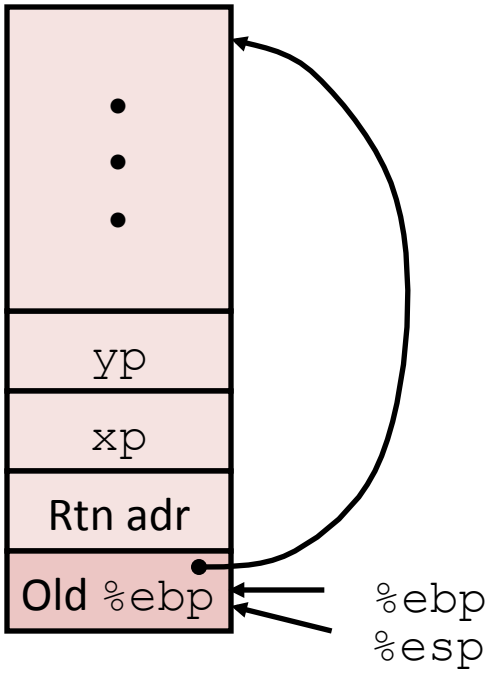

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack



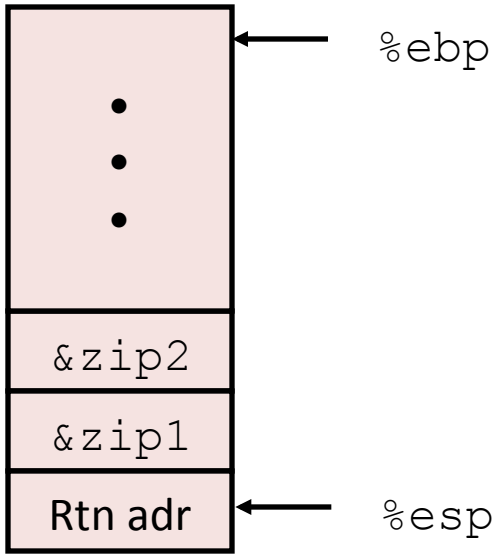
Resulting Stack

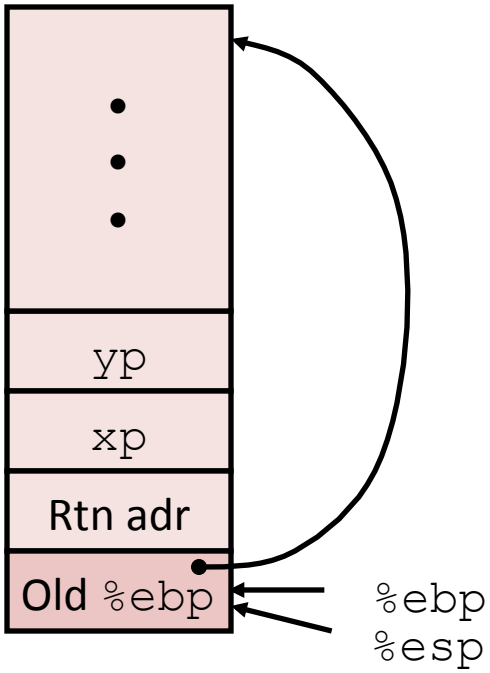



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

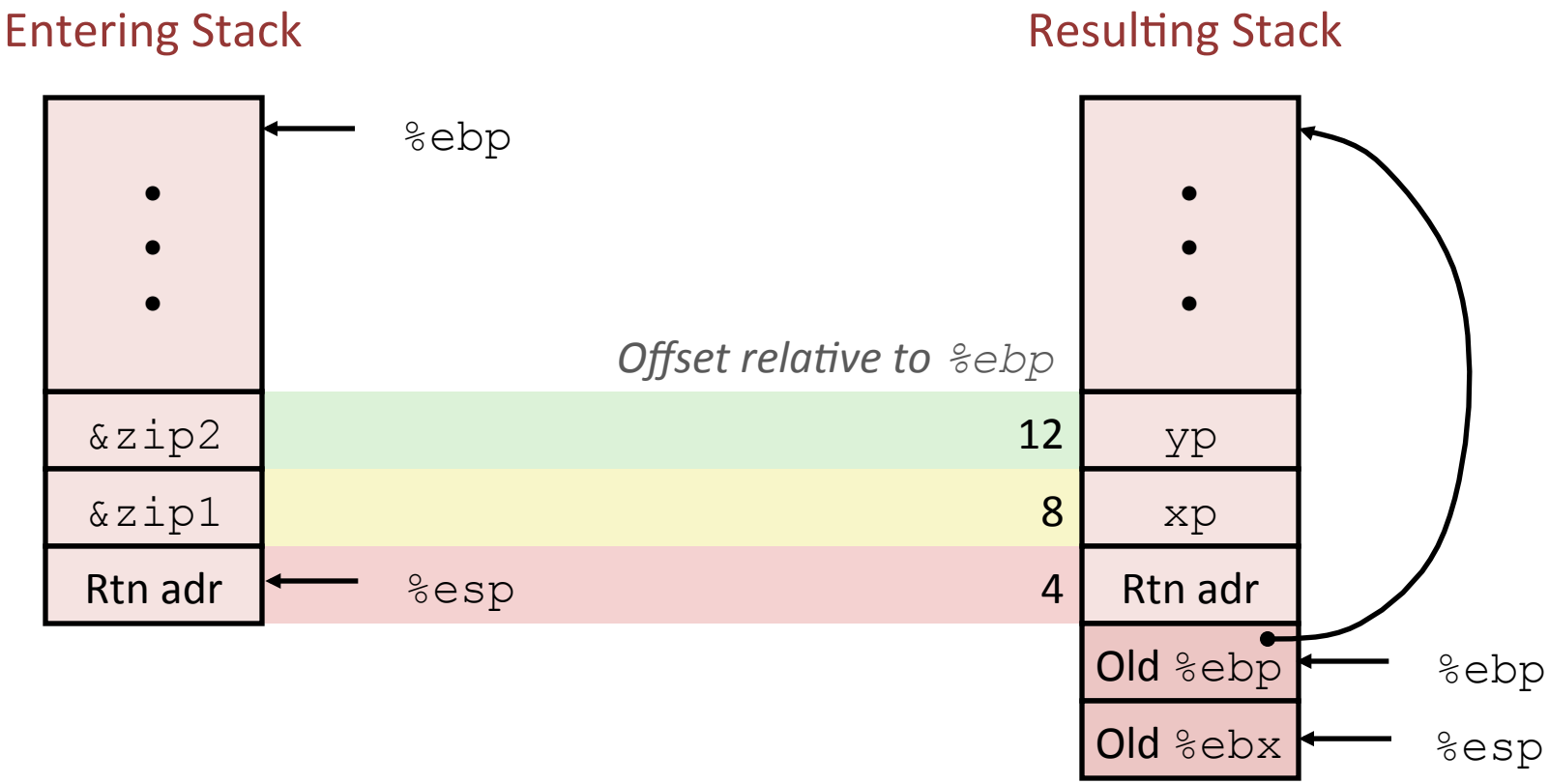
Entering Stack






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

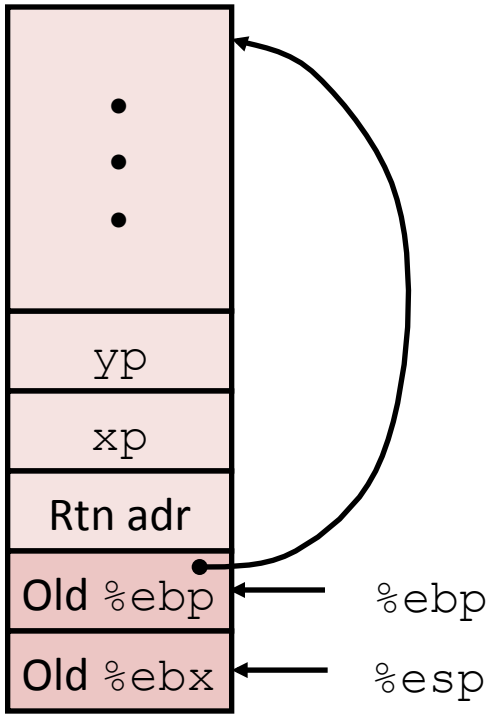
swap Setup #1



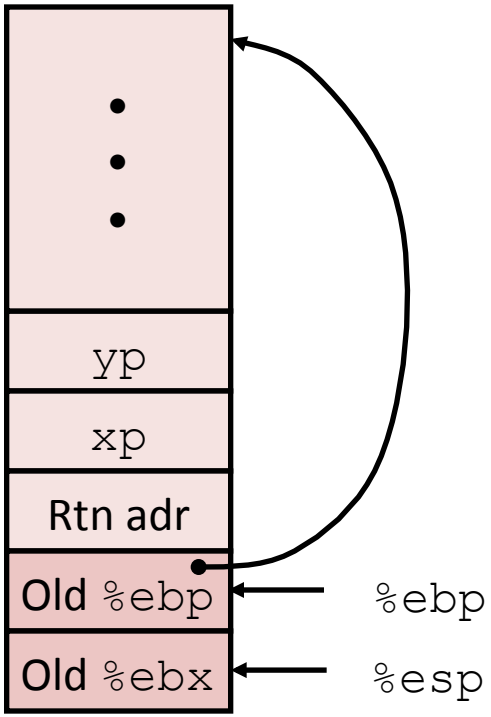

```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

swap Finish #1

swap's Stack



Resulting Stack

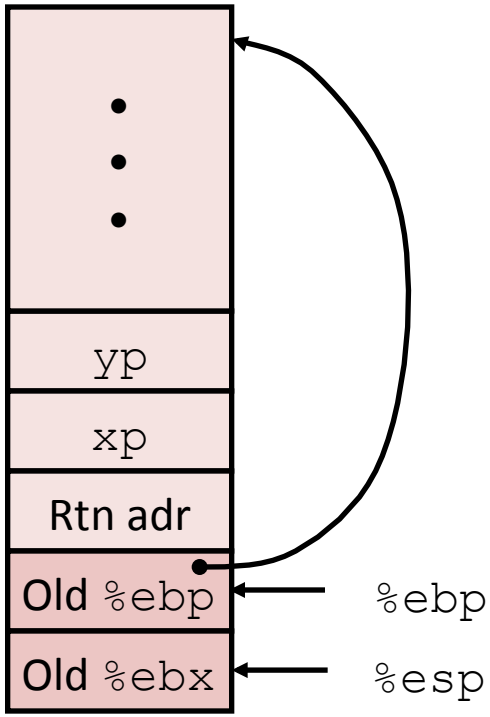


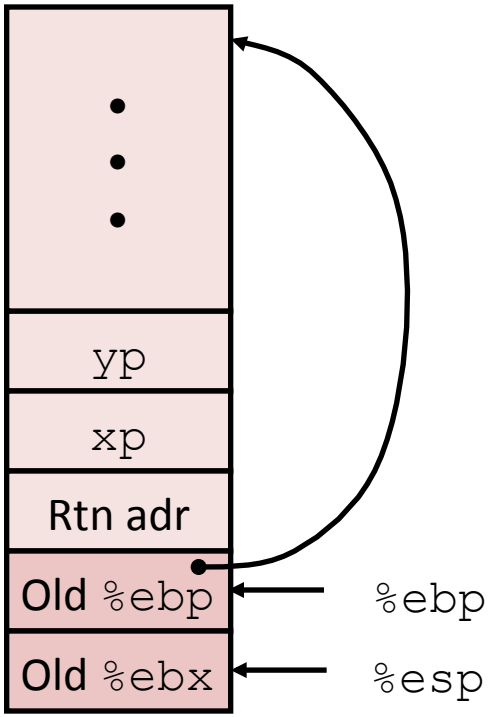

```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

swap Finish #2

swap's Stack

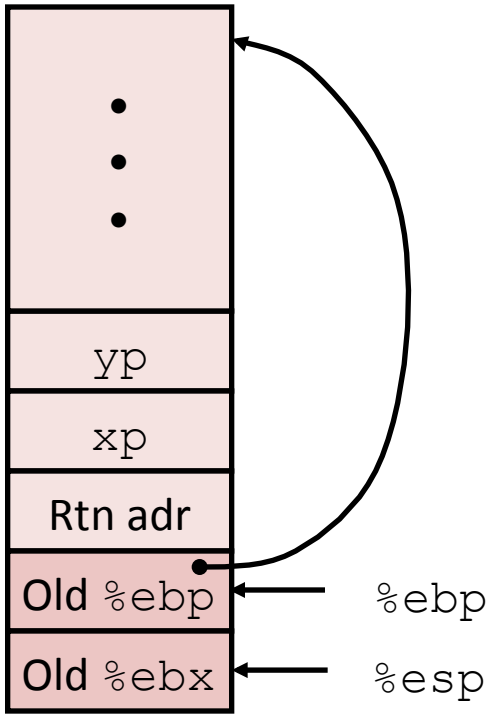




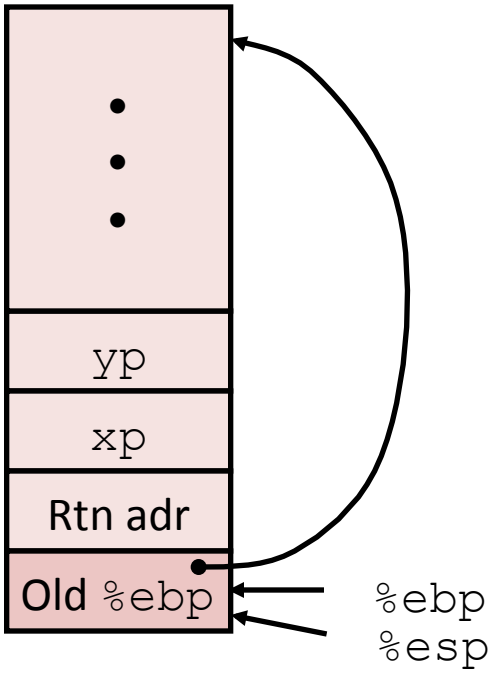

```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

swap Finish #2

swap's Stack



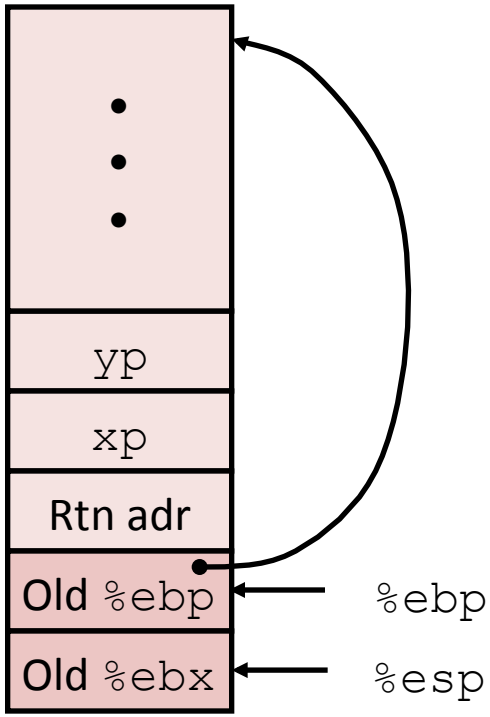
Resulting Stack

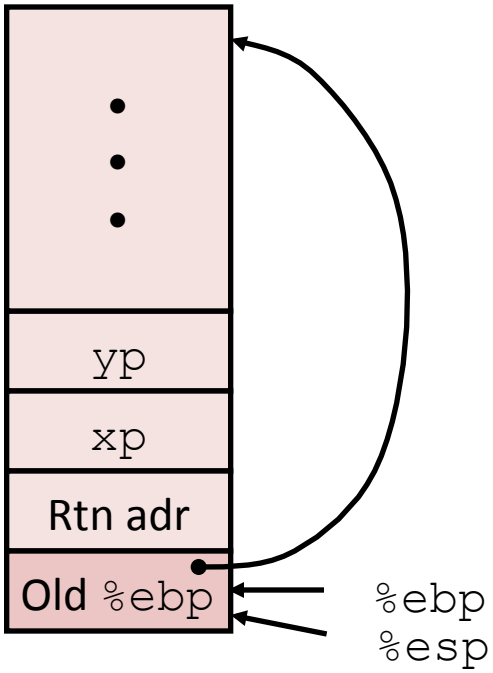



```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

swap Finish #2

swap's Stack

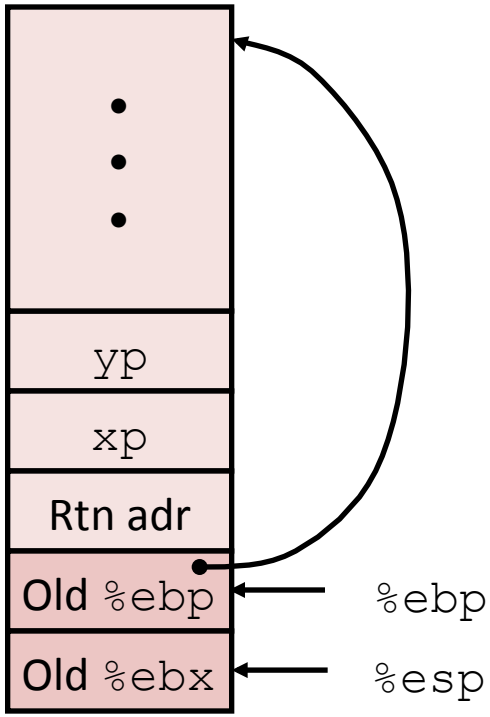




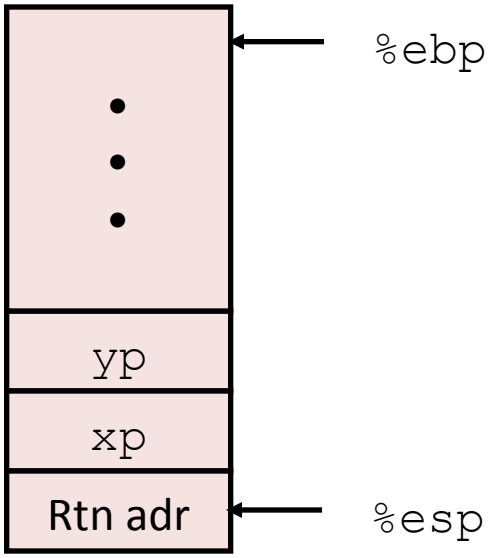

```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

swap Finish #3

swap's Stack



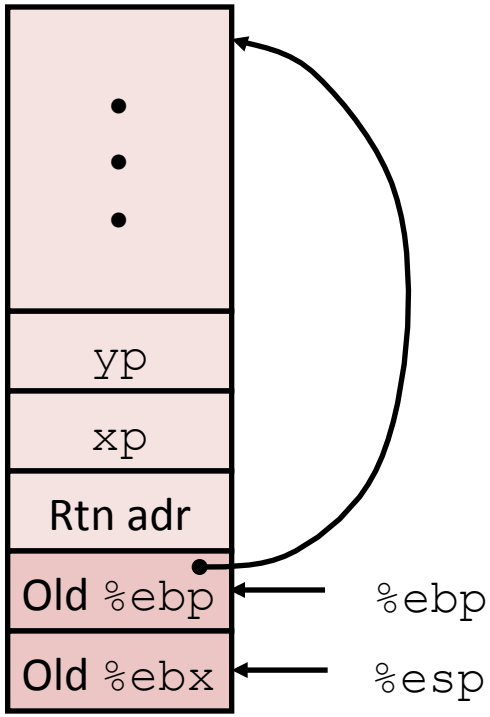
Resulting Stack

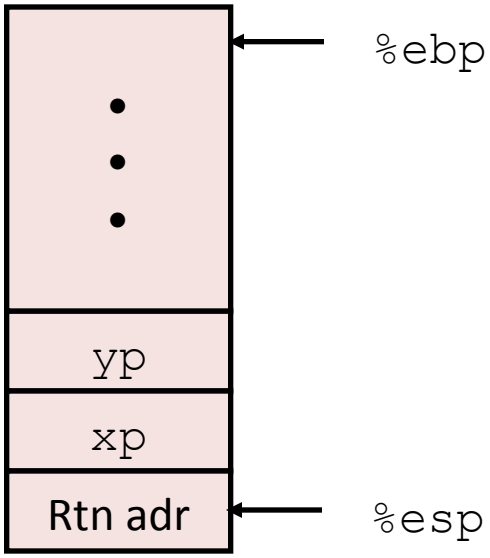



```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

swap Finish #4

swap's Stack

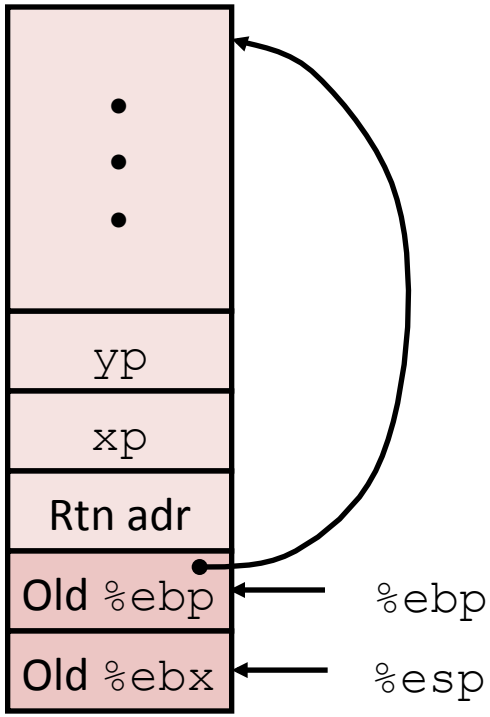




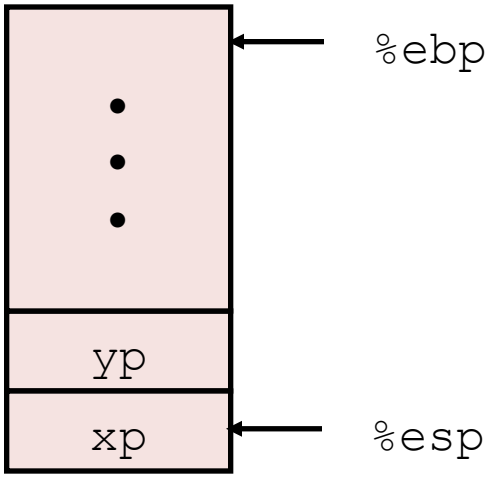

```
movl -4(%ebp),%ebx
movl %esp,%ebp
popl %ebp
ret
```

swap Finish #4

swap's Stack



Resulting Stack




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

```
movl %esp,%ebp
popl %ebp
```

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`**

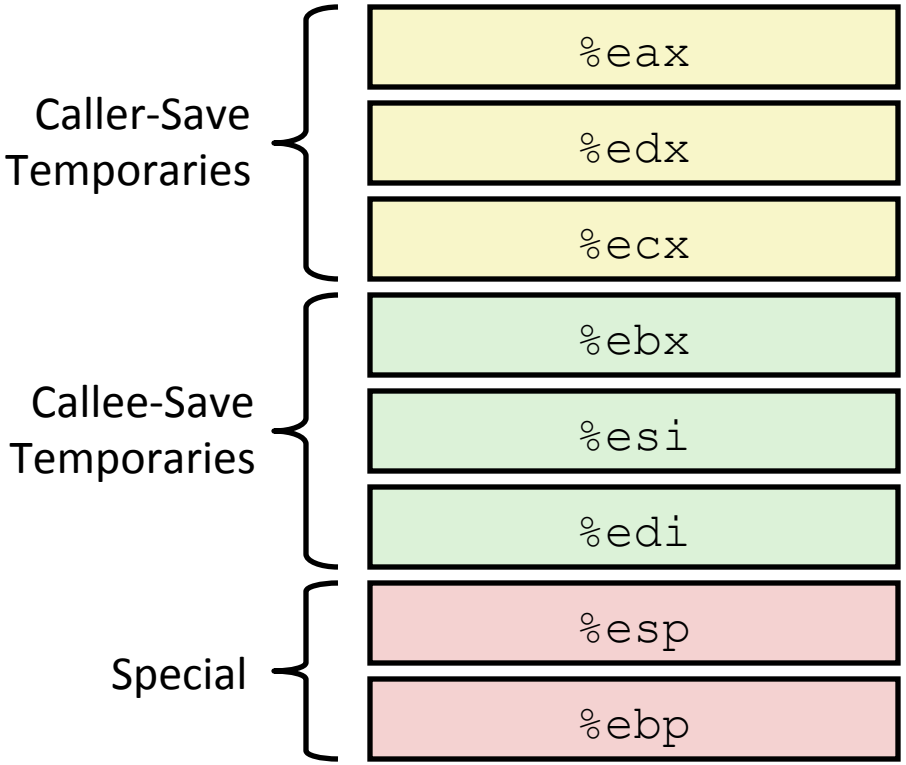
Register Saving Conventions

- 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



- When is caller save better?
  - What about when calling *multiple* functions?

- Returning floating point vs. returning int
  - Int: %eax (by default – not necessary)
  - Float: x87 register ST0