CSE 523S: Systems Security

Computer & Network Systems Security

Spring 2018

Jon Shidal

Plan for Today

- Announcements
  - I suggest you start the Python tutorial early
- Security news?
- CSE361 vs CSE523
- System Design & Security
  - Why are our computer systems vulnerable?
- Assignment: Reading and Python

Notes about CSE361 and CSE523

- CSE361 recently made the complete switch to x86-64 from IA32
- Today's CSE523 lecture looks at IA32
- CSE523 lectures will move entirely to x86-64 in the future
- But not this semester…
- We will at different times use both...

Notes about CSE361 and CSE523

- Where are the major differences
  - address sizes 32-bit vs. 64-bit
  - number of registers
  - register names: e.g. esp vs. rsp
  - argument passing to procedures: stack vs. regs
- Some links to CSE361 lectures on these areas:
  - Machine Basics (including Registers)
  - Control (jumps, branches, etc.)
  - Procedures
- This will be a brisk review! You should revisit these slides yourself as needed.

WHY ARE OUR COMPUTER SYSTEMS VULNERABLE?

Computers are Vulnerable

- Because we **write our own <u>software</u>**
  - Did we mistakenly/intentionally add vulnerabilities?

- Because we **choose our own <u>software</u>**
  - Can we know if it has vulnerabilities?

- Because **<u>software</u> requires input**
  - Can inputs be used to trigger a vulnerability?

How Vulnerable Is it?

- Write SW?
- Choose SW?
- Provide input?

How can I execute my code on your system?

- I can give you the program, and have you execute it for me
  - Ex: *Email: Please download and run this attachment*
  - *Ex: Verisign mistake*
- I can gain access to your machine and execute it myself
  - Ex: Exploit a system vulnerability to gain access
  - Ex: Steal credentials to gain access

Let's review how code gets executed

- Adopt this mindset
  - We write our code into memory, and give a starting address to the CPU
  - The CPU executes a simple machine language
  - Assembly code is nothing to fear
- We will be looking at binaries throughout the semester, so let's start from the beginning
- Book uses Intel assembly syntax, our slides use AT&T syntax. comparison of the two

Intel “x86” Processors

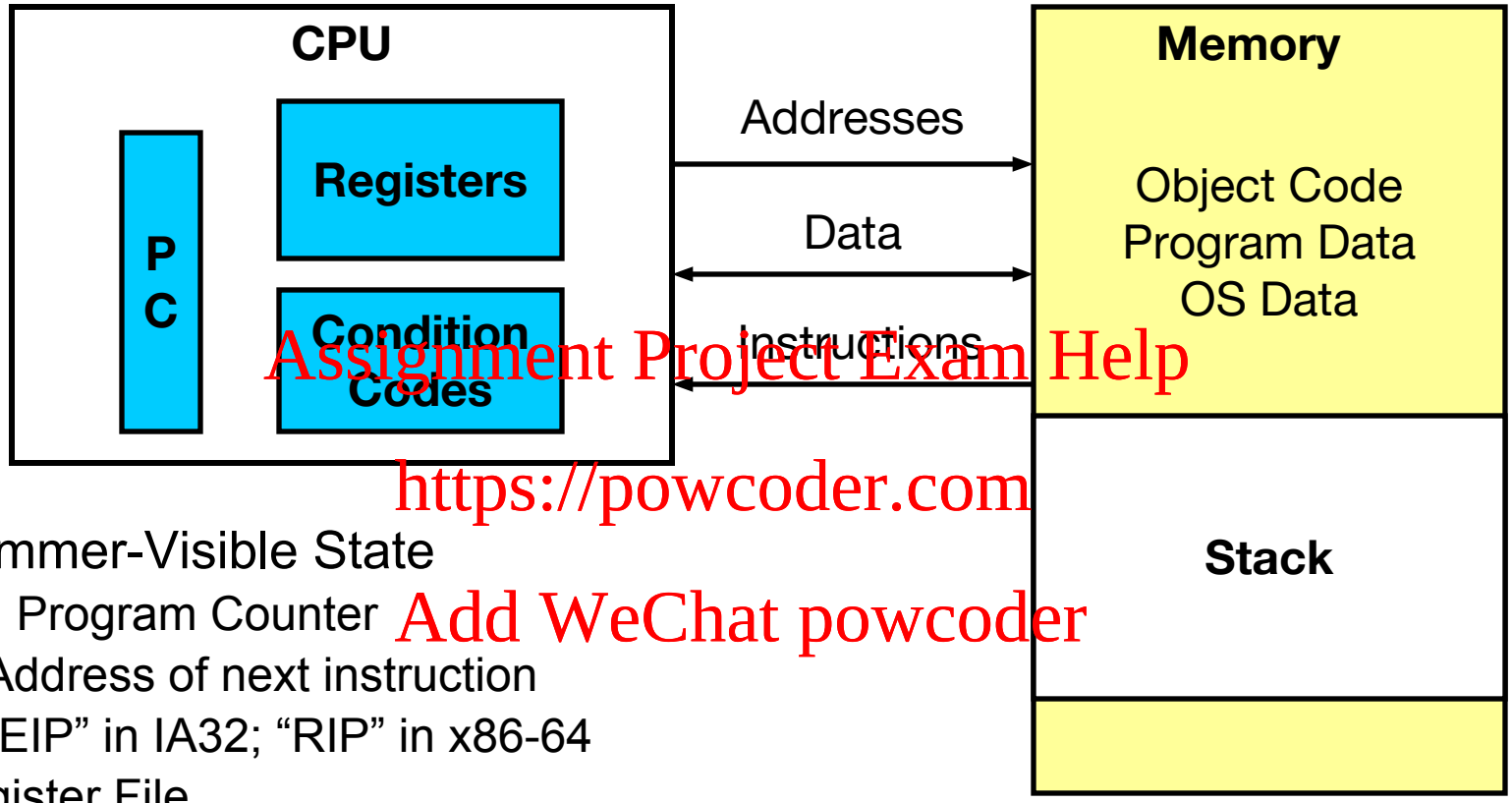
Many of following slides taken from CSE 361, based on Computer Systems, by Bryant & O’Hallaron

- Dominate Computer Market

- Evolutionary Design
  - Starting in 1978 with 8086
  - Add more features as time goes on
  - Still support old features, although obsolete

- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

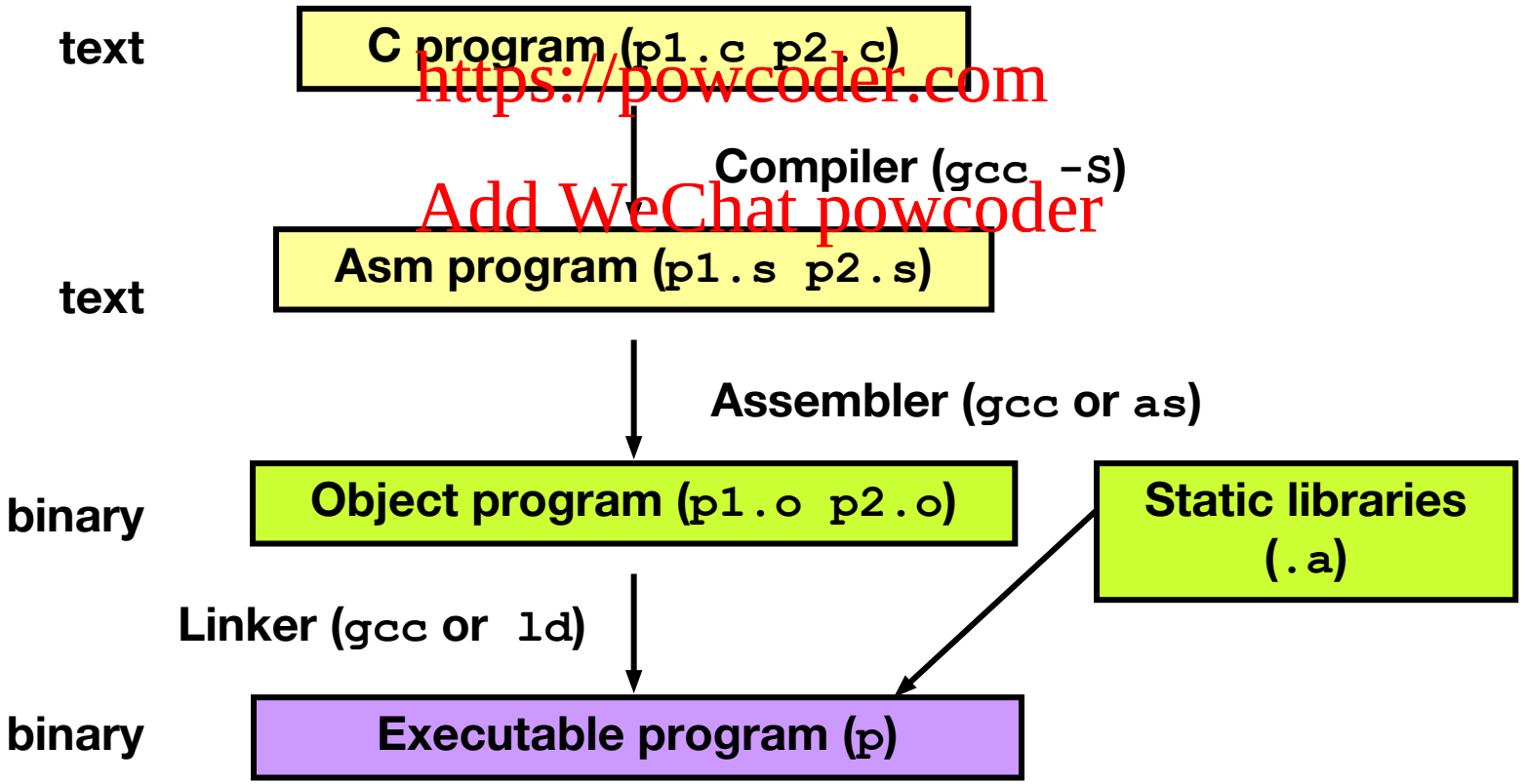
Assembly Programmer's View



- Programmer-Visible State
  - PC: Program Counter
    - Address of next instruction
    - "EIP" in IA32; "RIP" in x86-64
  - Register File
    - Heavily used program data
  - Condition Codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
  - Memory
    - Byte addressable array
    - Code, user data, (some) OS data
    - Includes stack used to support procedures

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```


Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s` **because of** `-S`

Using -O will produce optimized results

Try and compare:

```
gcc -S code.c
```

Are we using 32-bit or 64-bit instructions?

Try -m32 and -m64 to see differences

One more thing: compilers change

Exact .s results might vary depending on version of gcc

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

Or

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046: 03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    `x`: Register `%eax`
    `y`: Memory M[`%ebp+8`]
    `t`: Register `%eax`
      - Return function value in `%eax`
- Object Code
  - 3-byte instruction
  - Stored at address `0x401046`

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:  55                    push   %ebp
   1:  89 e5                 mov    %esp,%ebp
   3:  8b 45 0c              mov    0xc(%ebp),%eax
   6:  03 45 08              add    0x8(%ebp),%eax
   9:  89 ec                 mov    %ebp,%esp
   b:  5d                    pop    %ebp
   c:  c3                    ret
   d:  8d 76 00              lea    0x0(%esi),%esi
```


- Disassembler

  `objdump -d p`

  – Useful tool for examining object code
  – Analyzes bit pattern of series of instructions
  – Produces approximate rendition of assembly code
  – Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly w/ gdb

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:  push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```


- Within gdb Debugger

  `gdb p`

  `disassemble sum`

  – Disassemble procedure

  `x/13b sum`

  – Examine the 13 bytes starting at `sum`

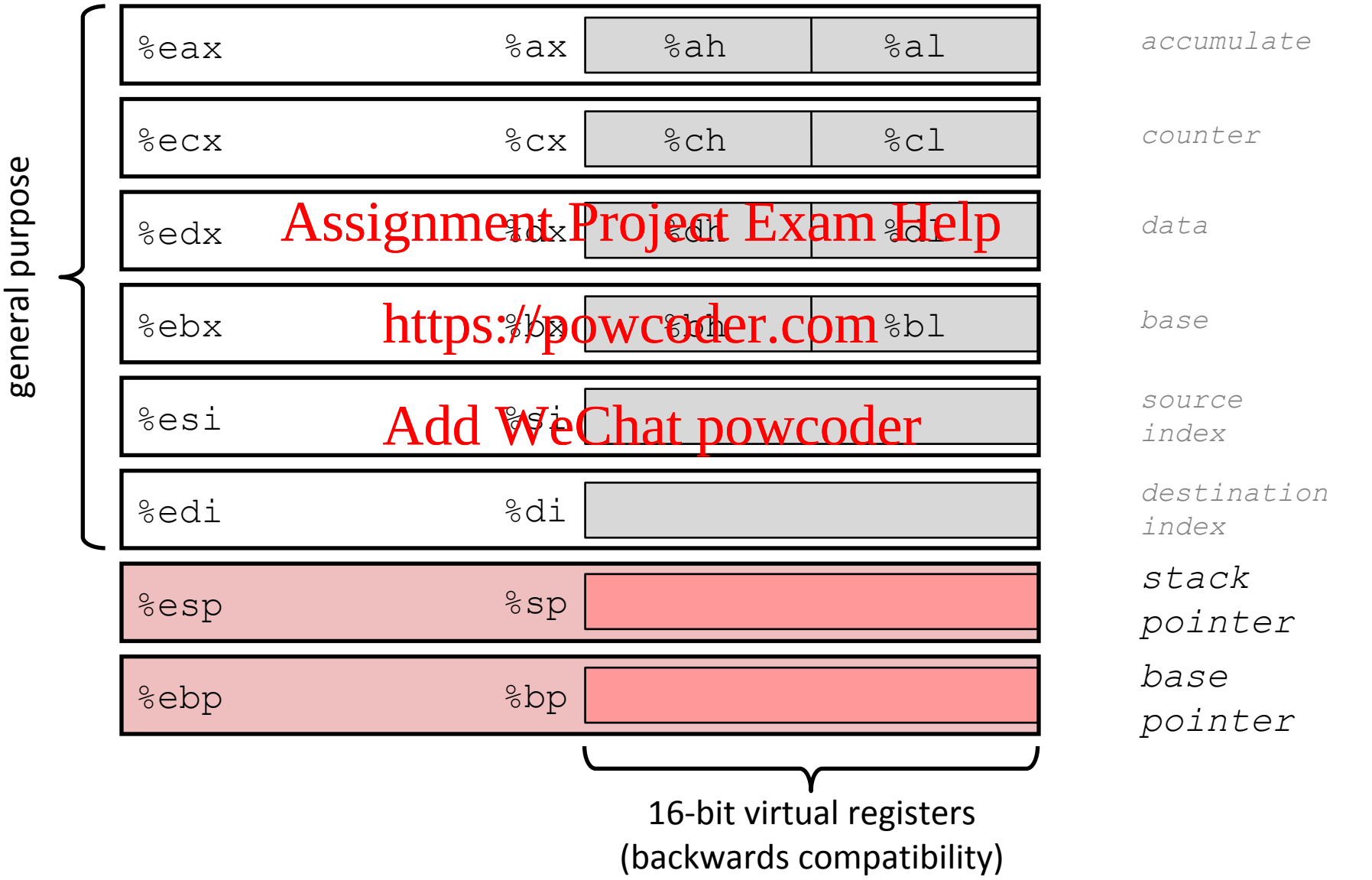
What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55                   push   %ebp
30001001: 8b ec                mov    %esp,%ebp
30001003: 6a ff                push   $0xffffffff
30001005: 68 90 10 00 30       push   $0x30001090
3000100a: 68 91 dc 4c 30       push   $0x304cdc91
```


- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

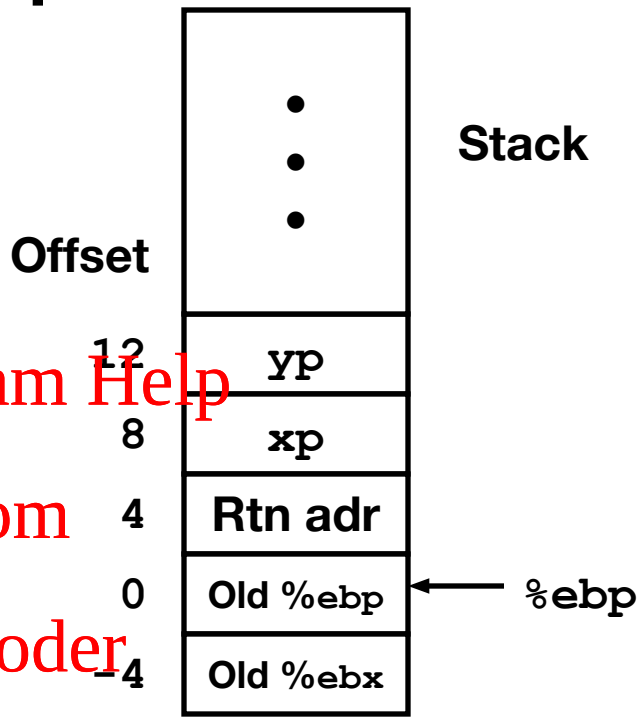
- BUT be careful, reverse engineering forbidden by Microsoft end user license agreement!

Integer Registers (IA32)



Understanding Swap

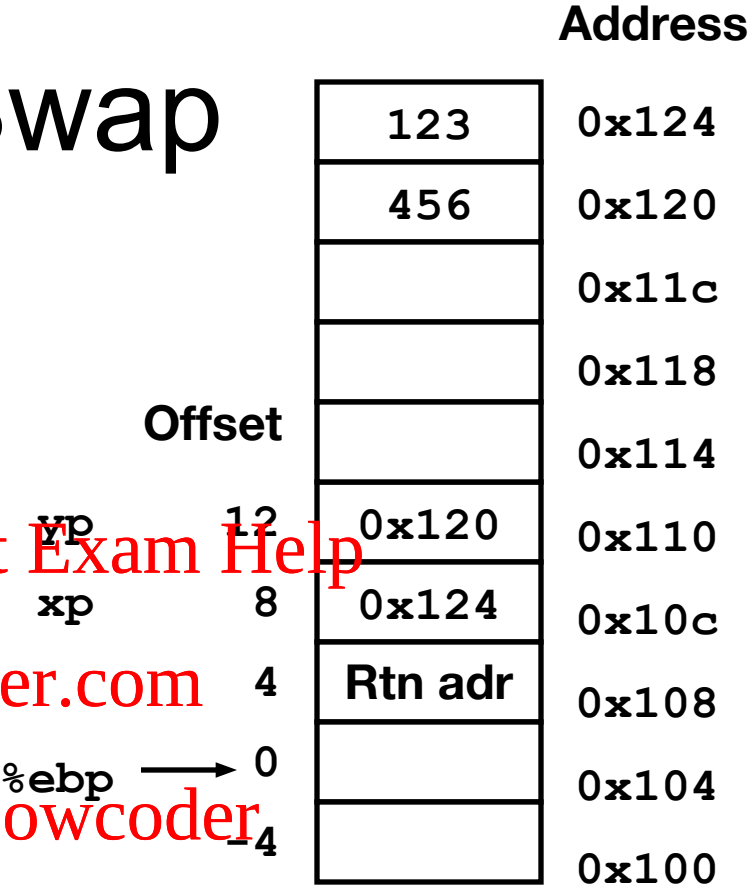
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

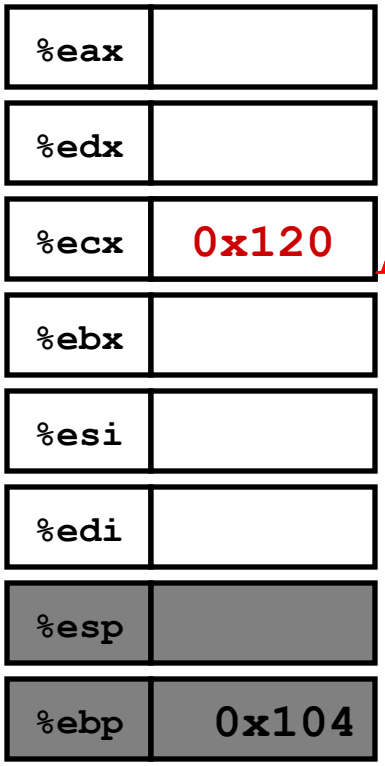
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx
```

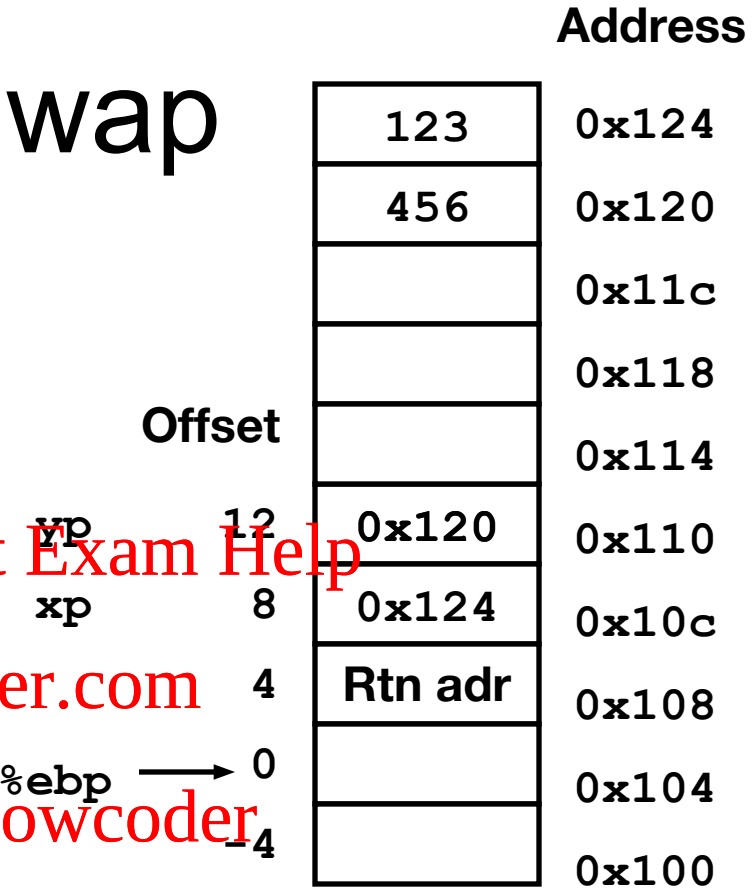
Understanding Swap




```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

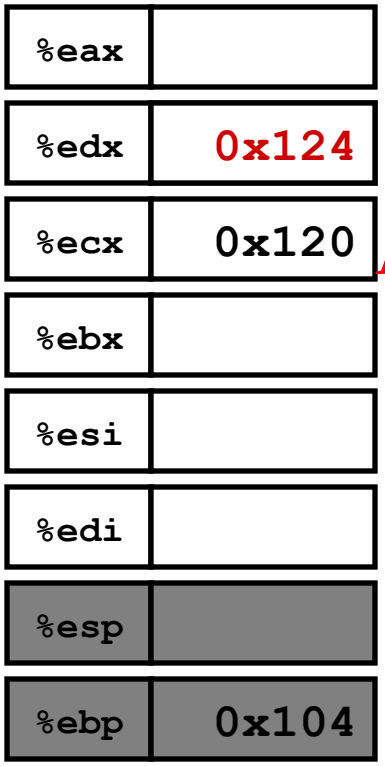
Understanding Swap

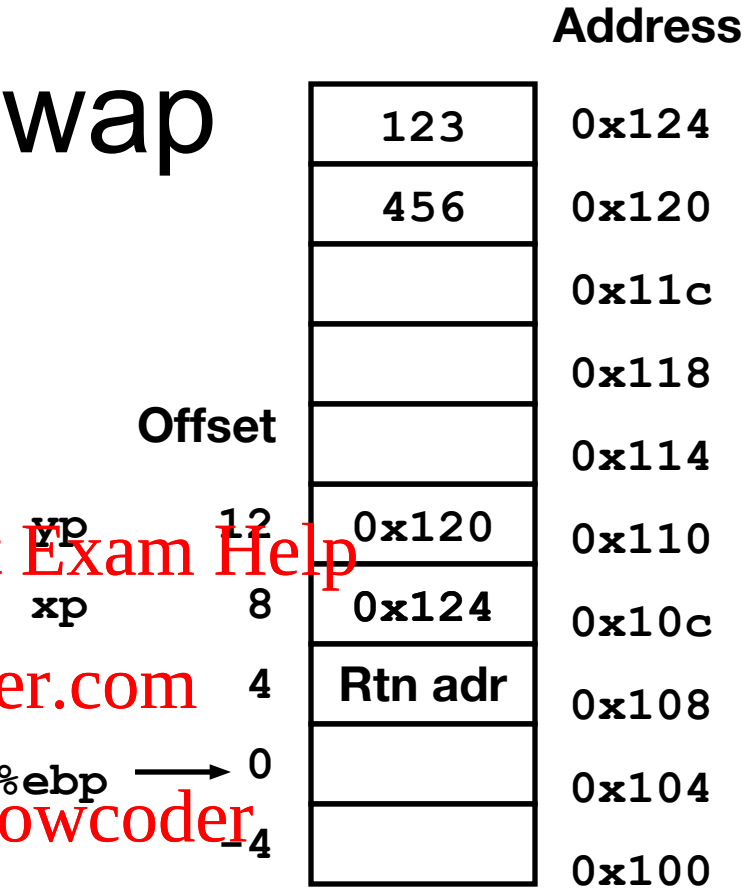





```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

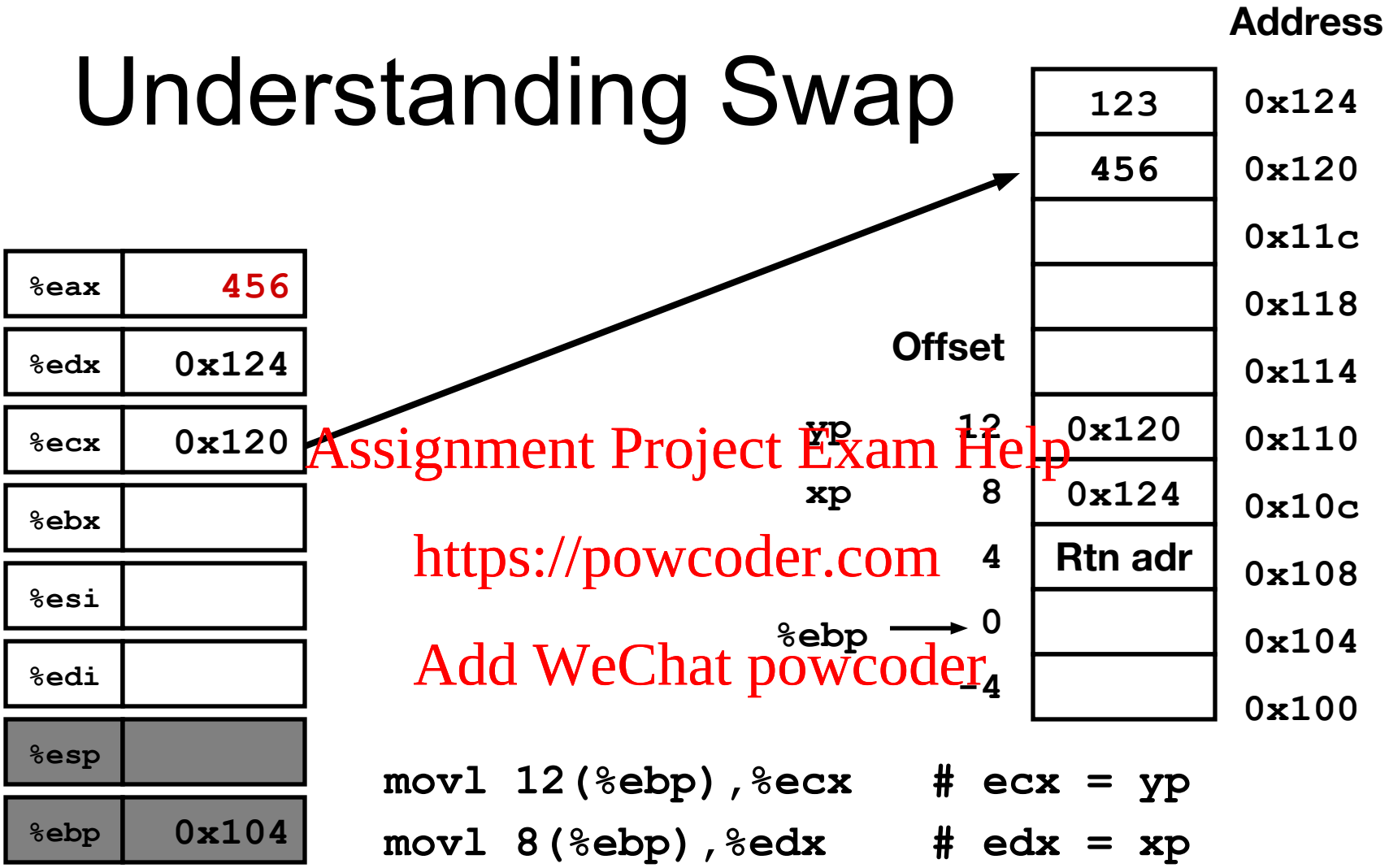
Understanding Swap





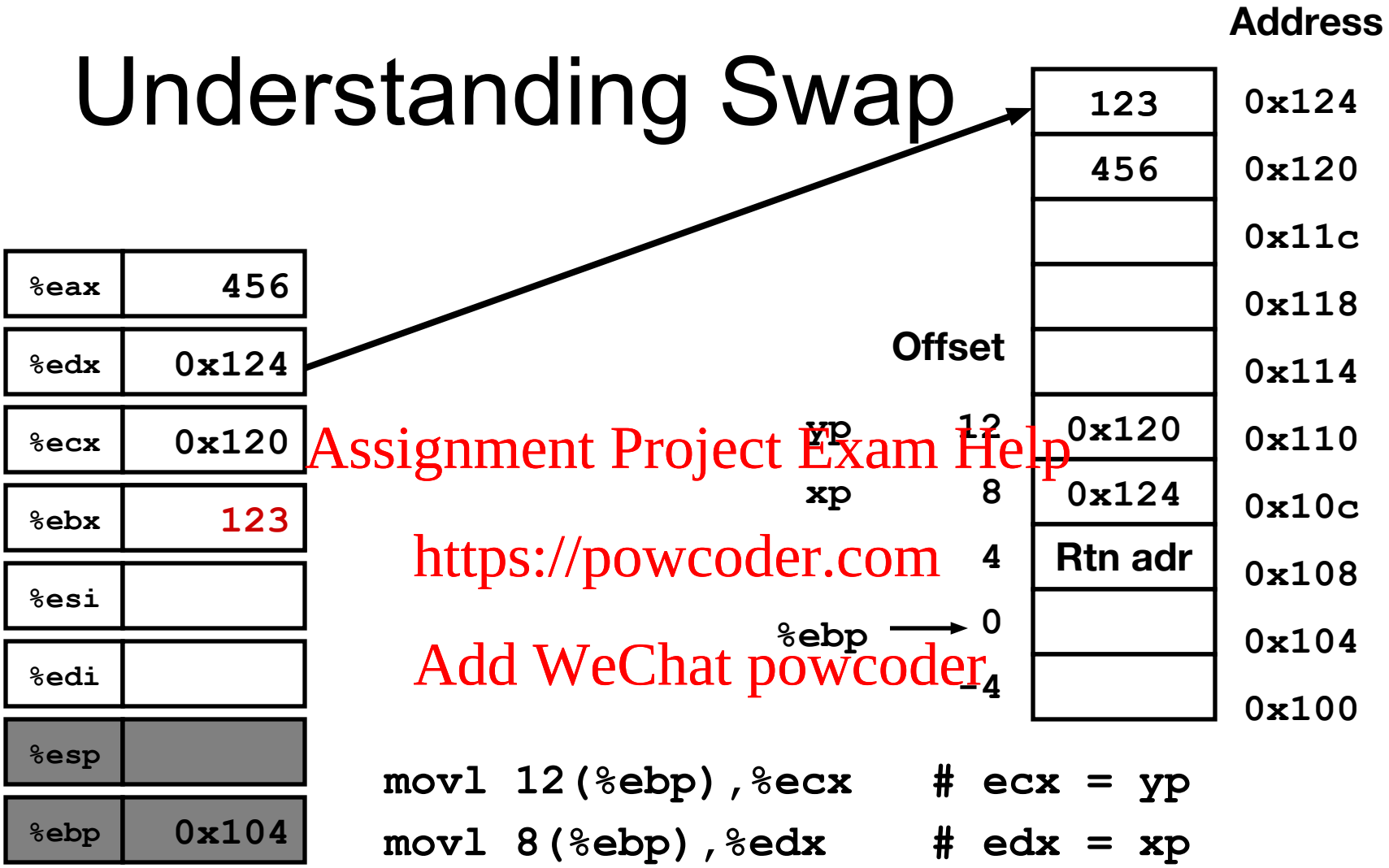

```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Understanding Swap



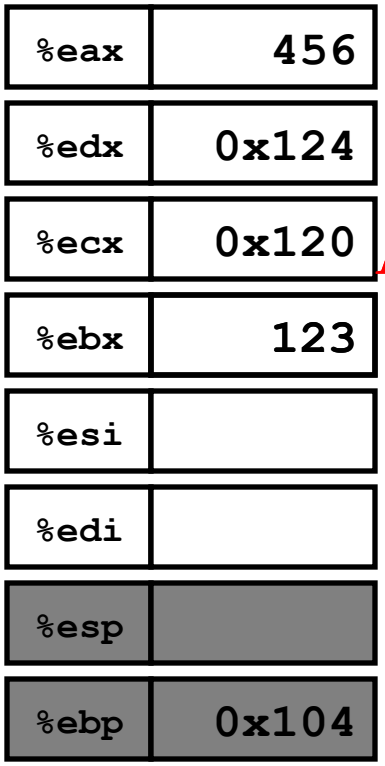

```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

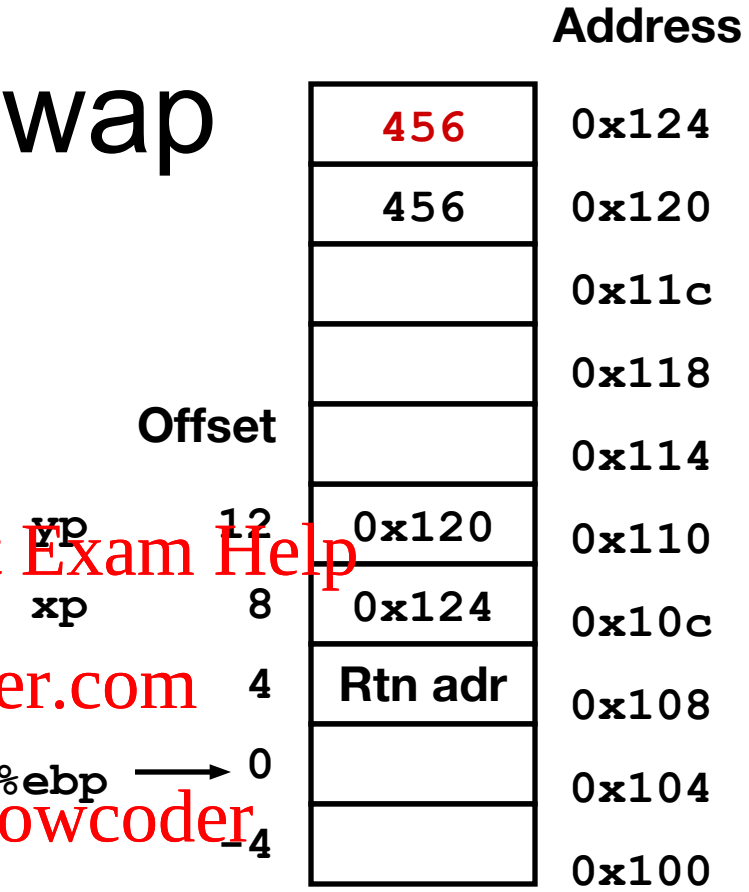
Understanding Swap




```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

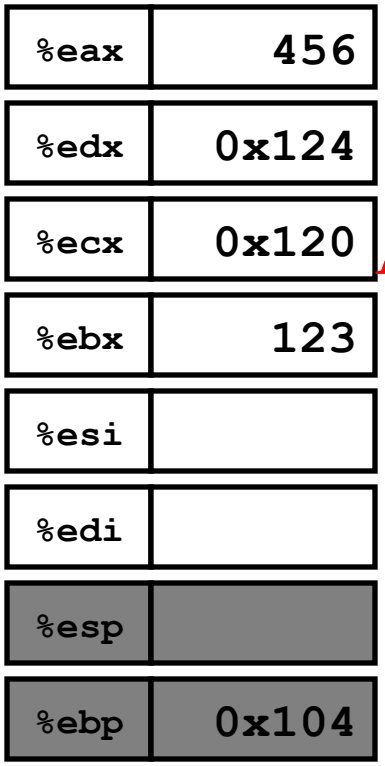
Understanding Swap

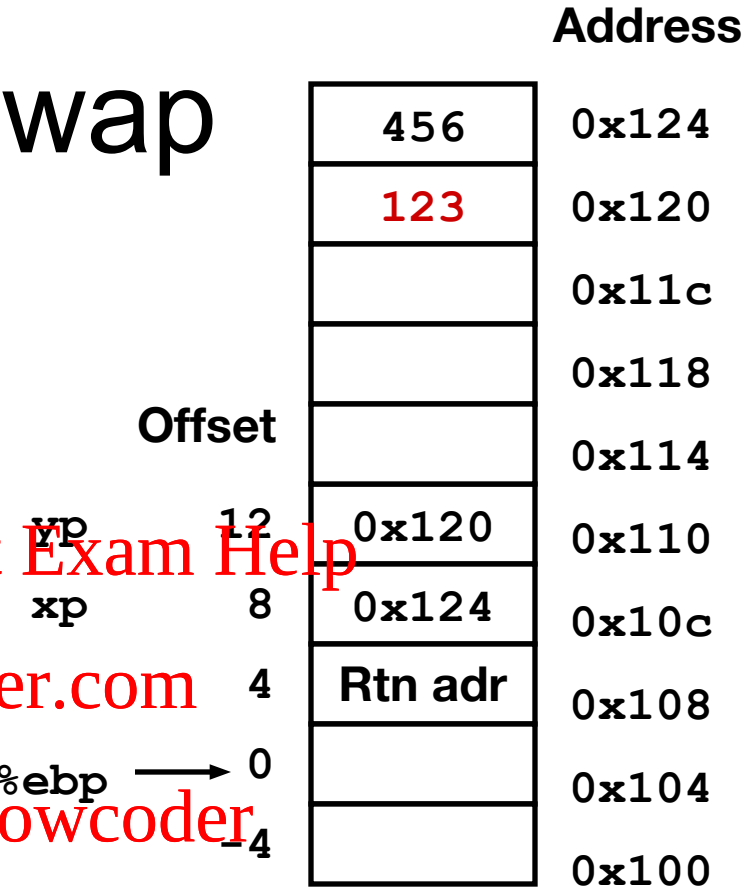





```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Understanding Swap

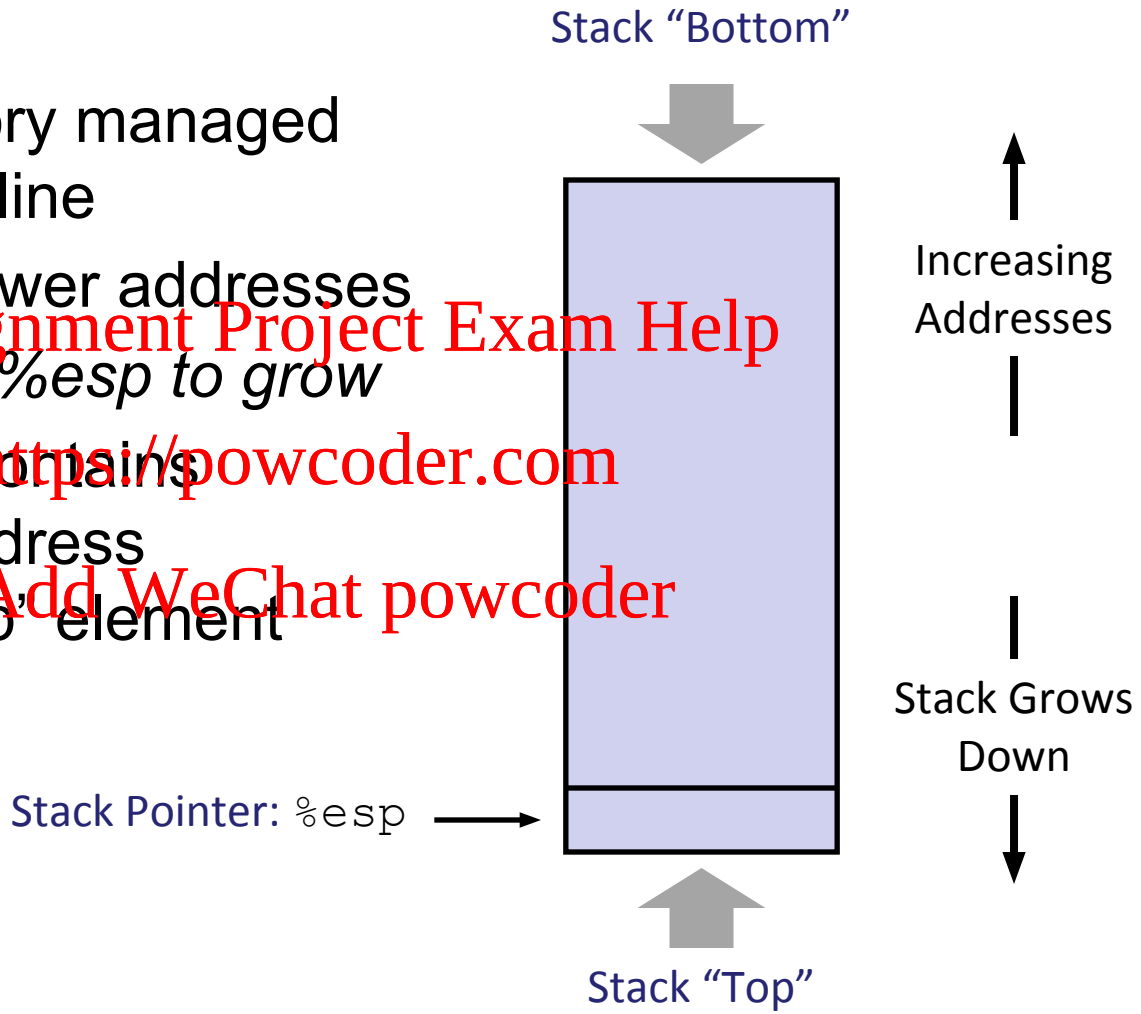





```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

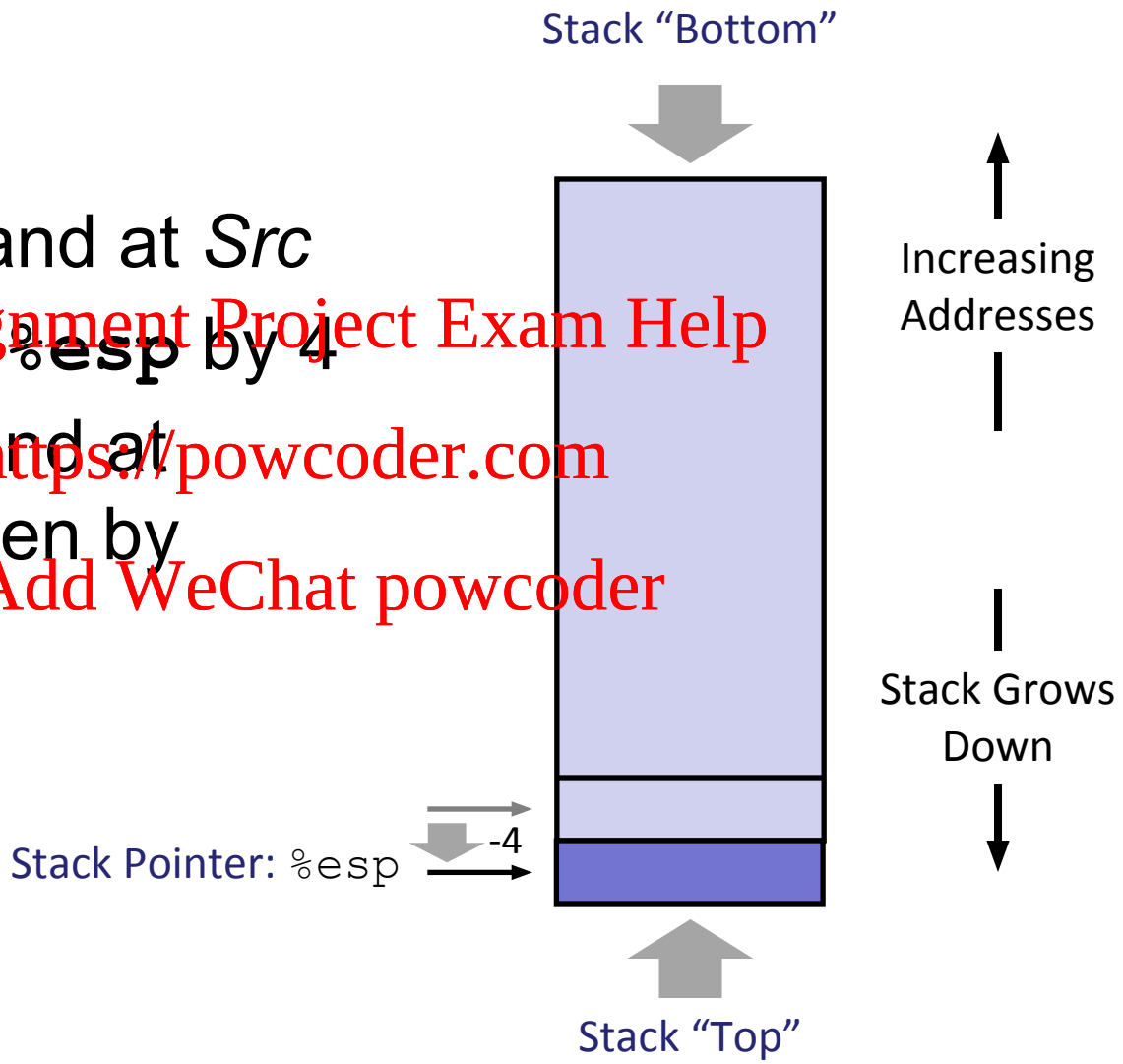
IA32 Stack

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



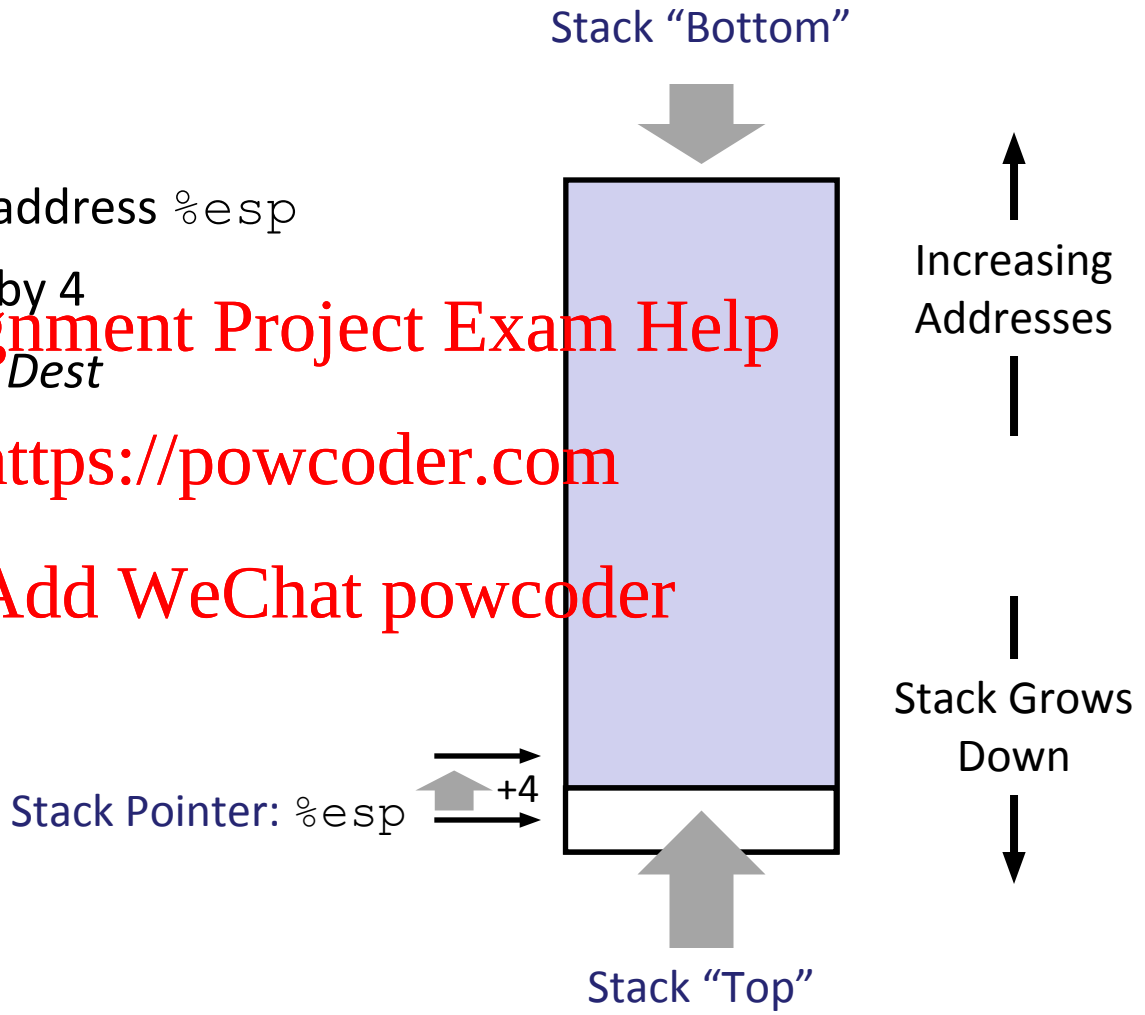
IA32 Stack: Push

- `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*



Procedure Control Flow

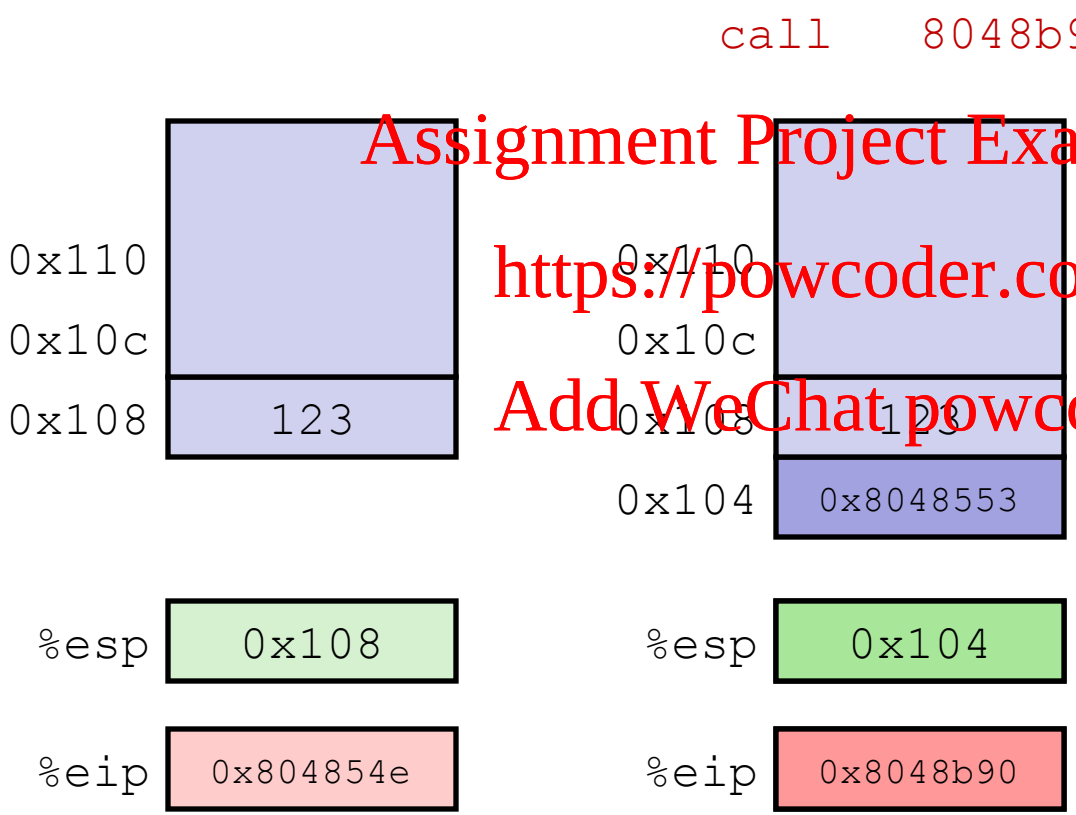
- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of instruction beyond **`call`**
  - Example from disassembly


```
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

Procedure Call Example

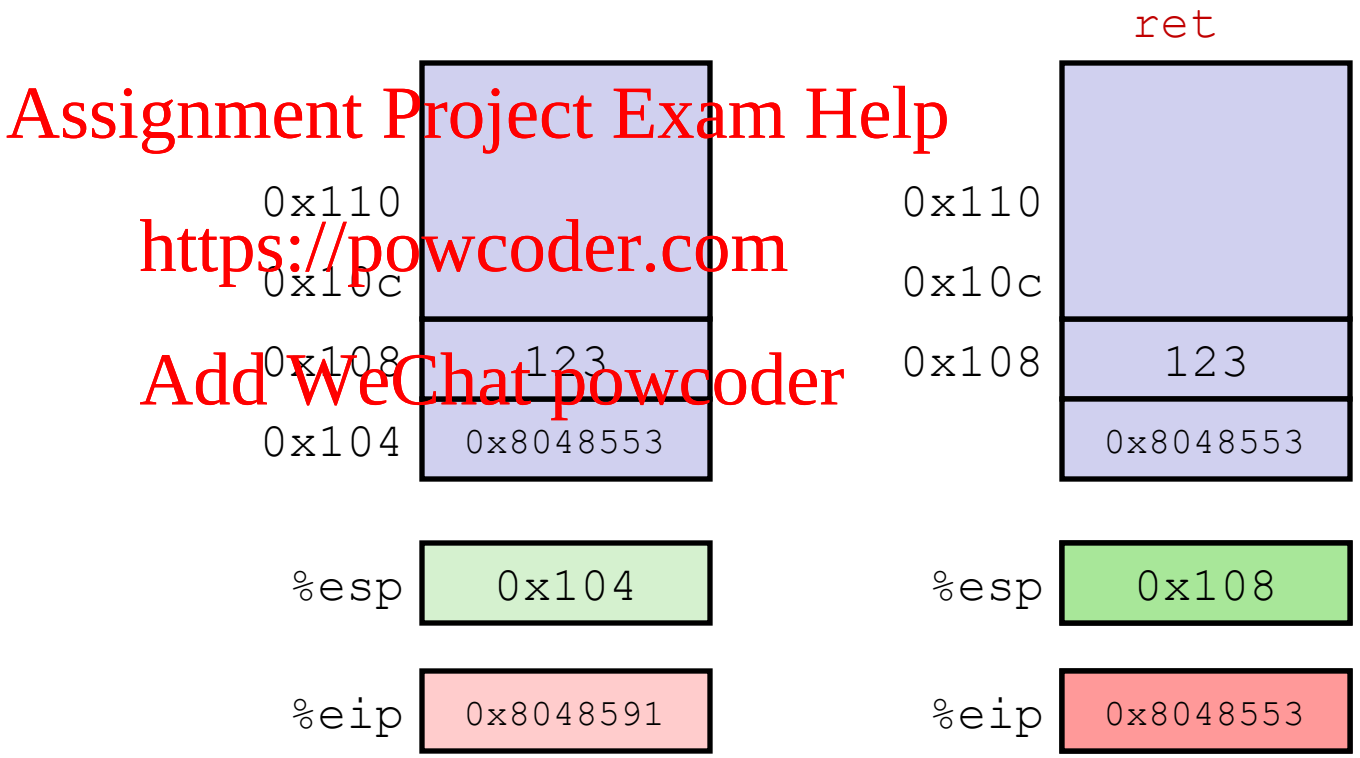
```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```



%eip: program counter

Procedure Return Example

```
8048591: c3                    ret
```



%eip: program counter

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, Postscript
  - 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

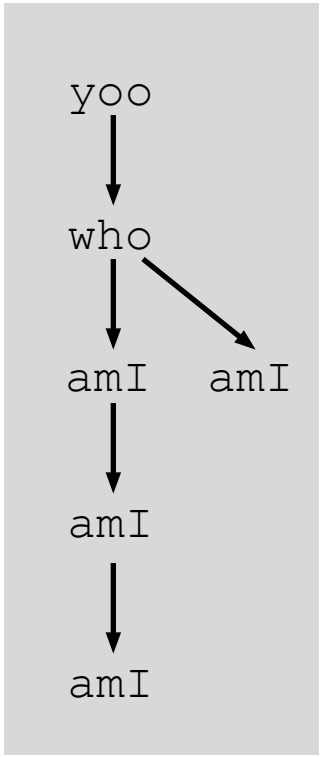
Call Chain Example

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

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

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

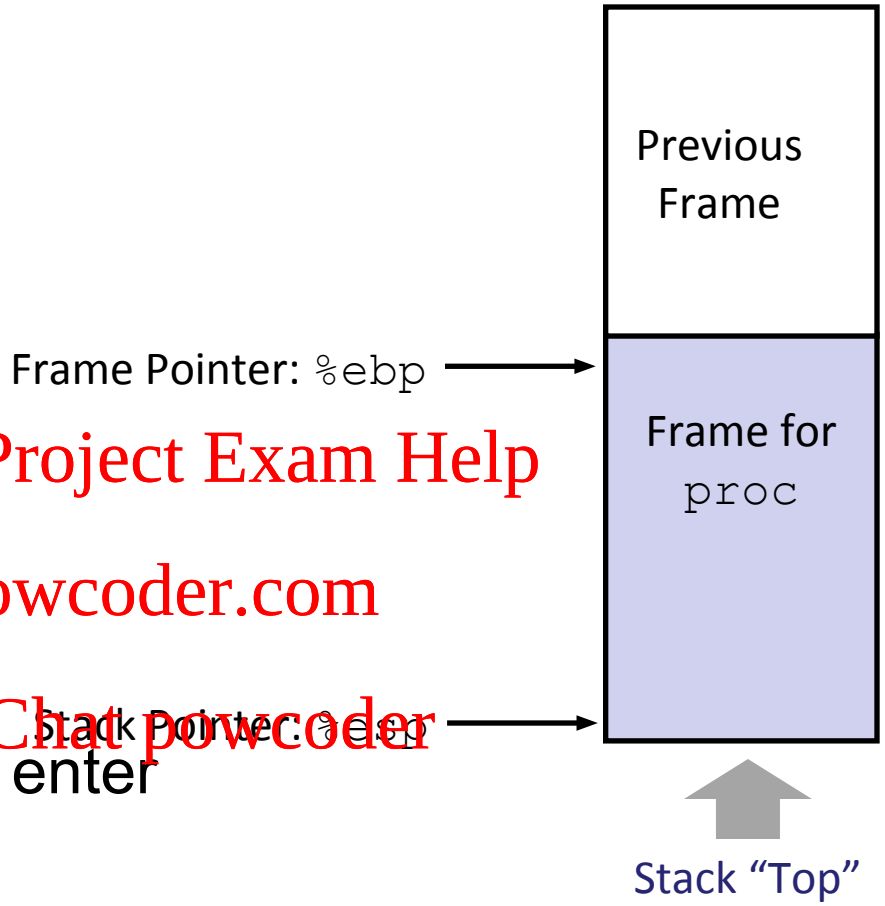
Example
Call Chain



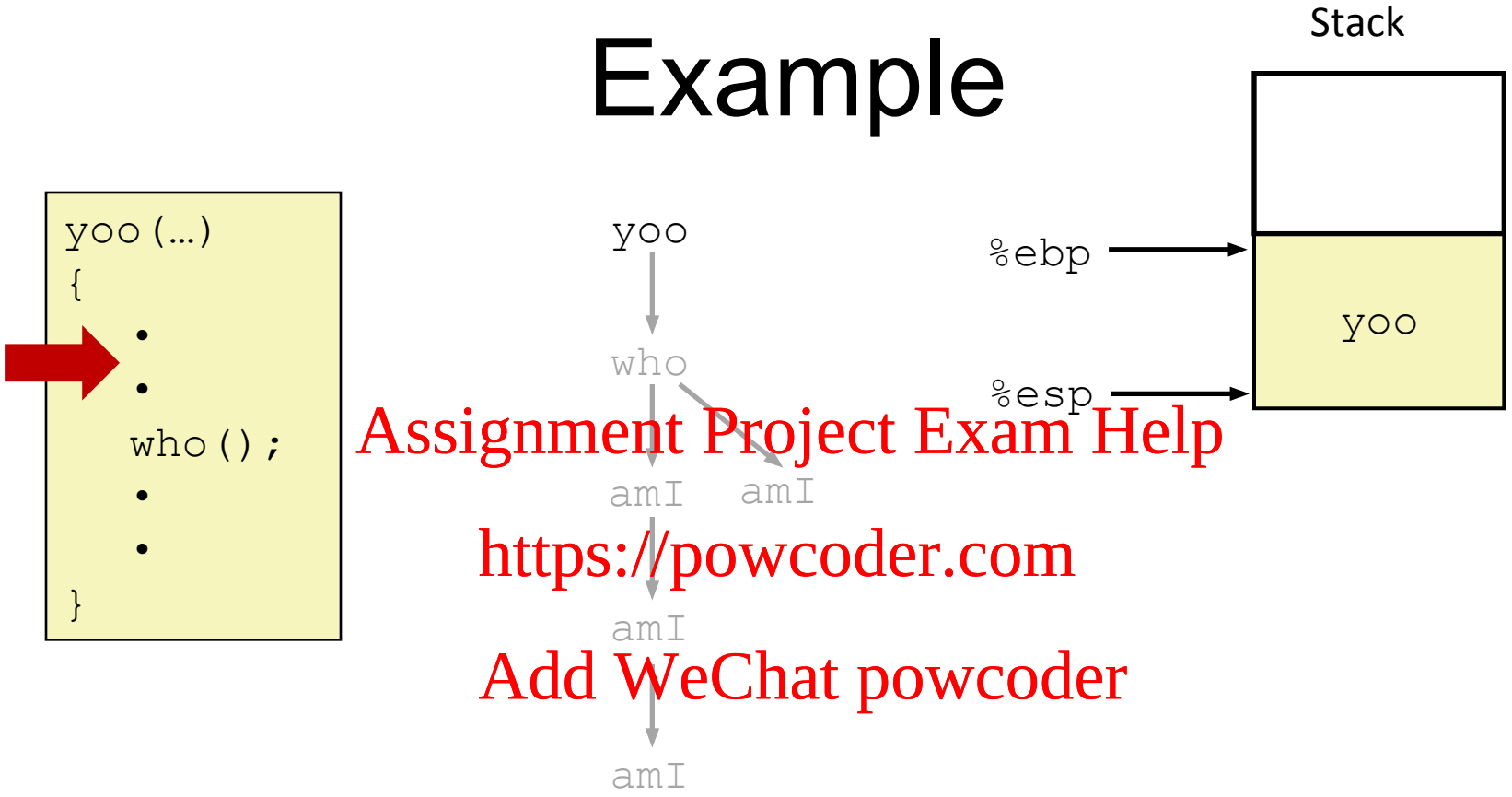
Procedure `amI` is recursive

Stack Frames

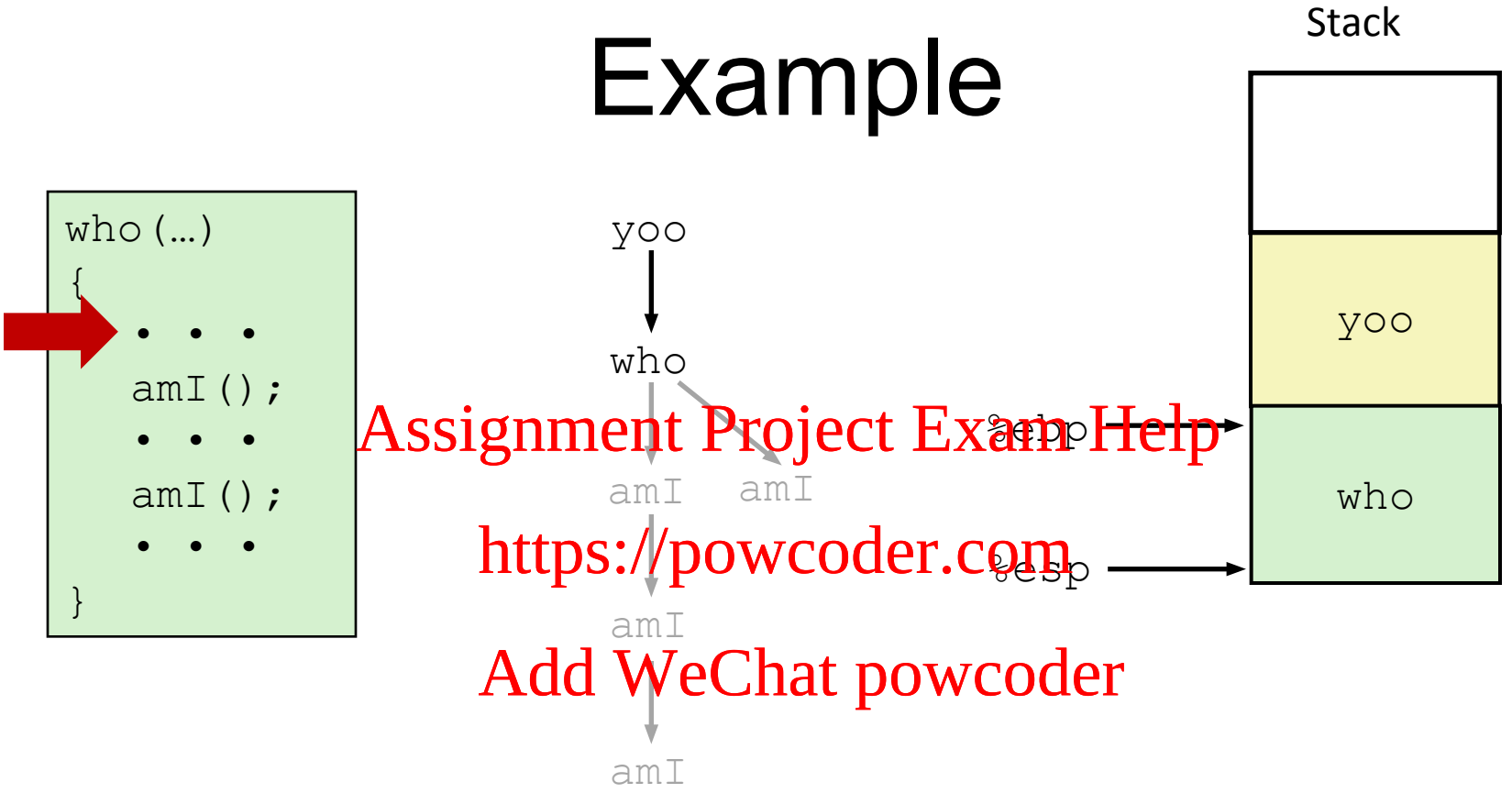
- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code



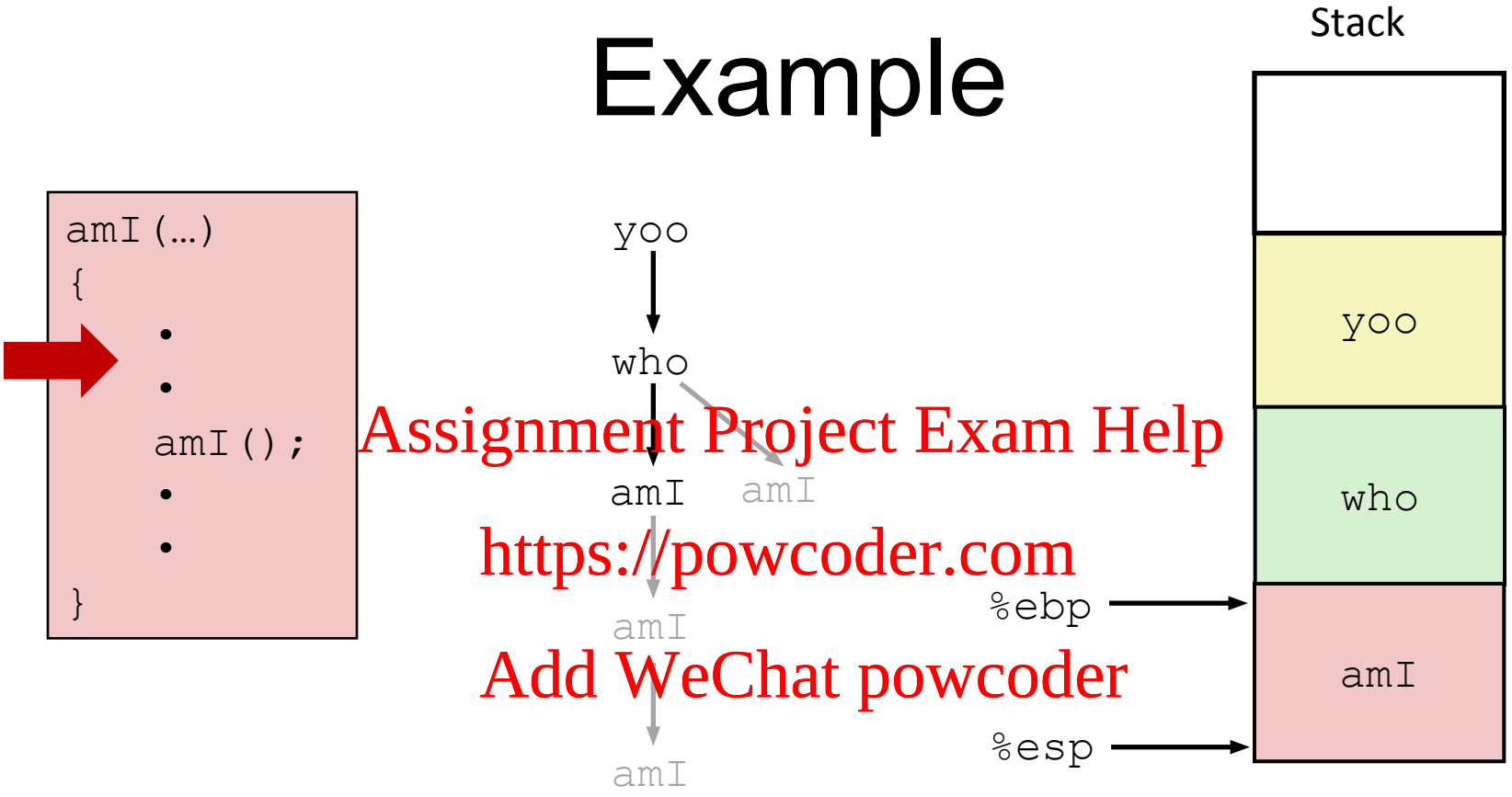
Example



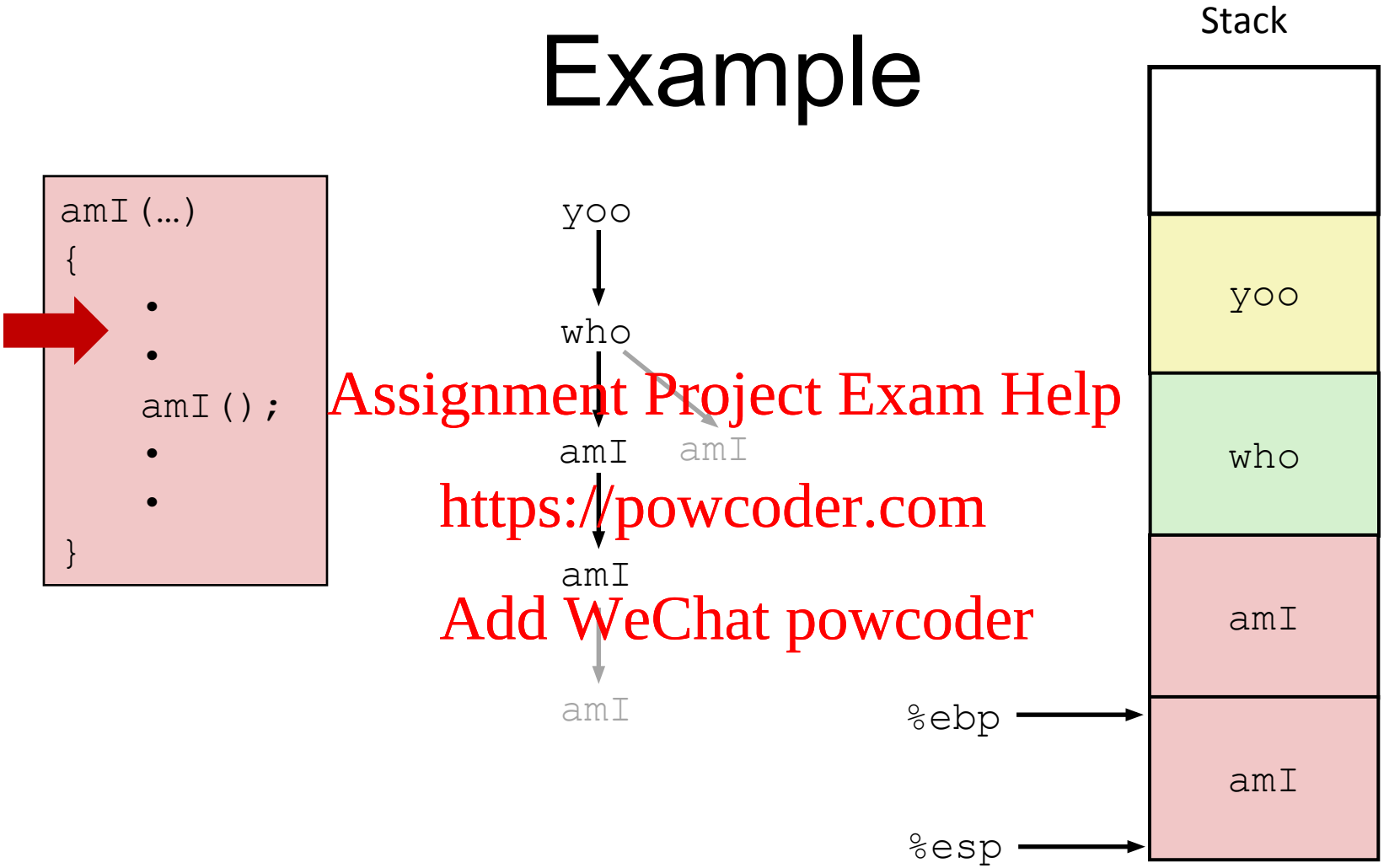
Example



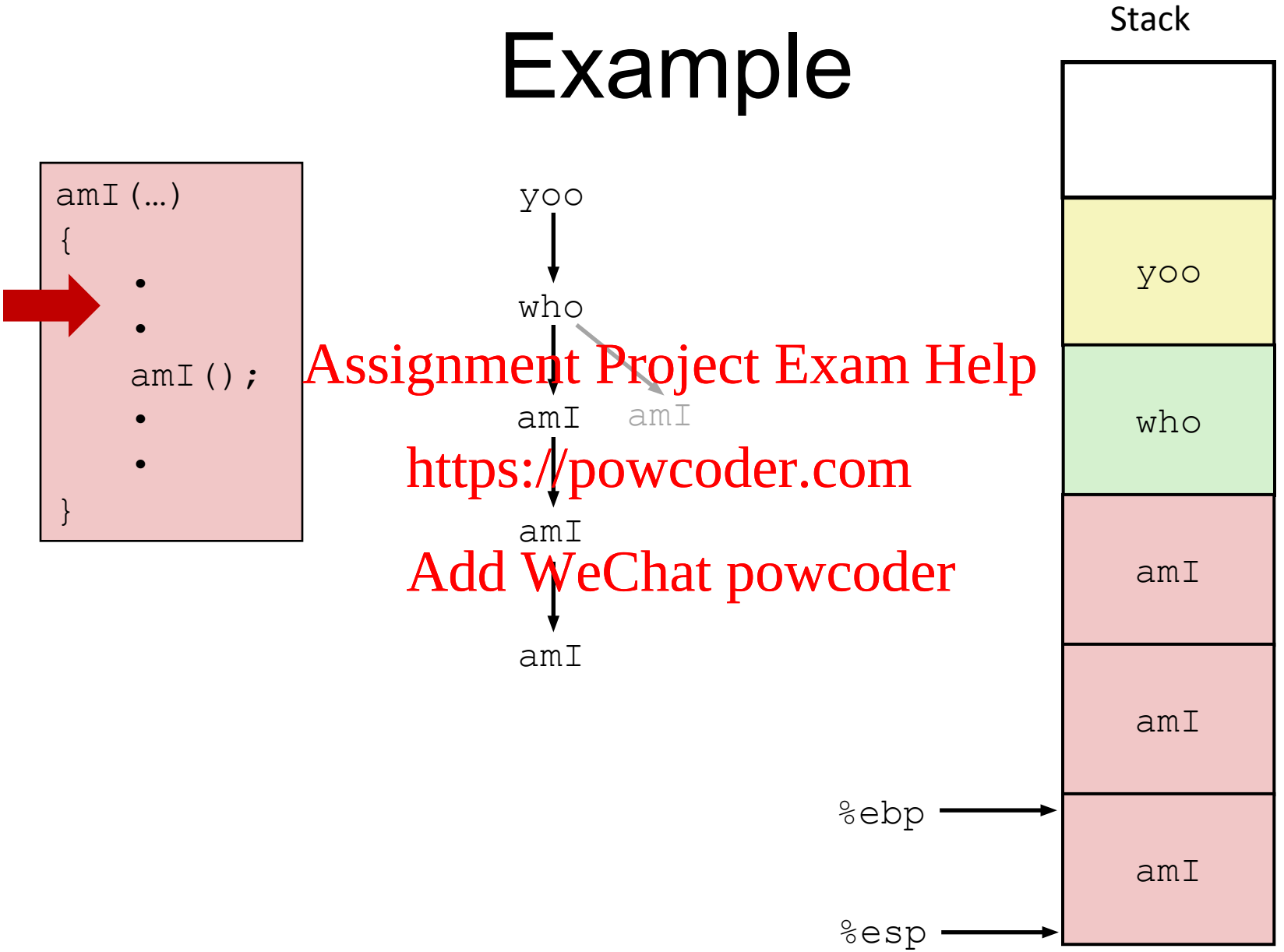
Example



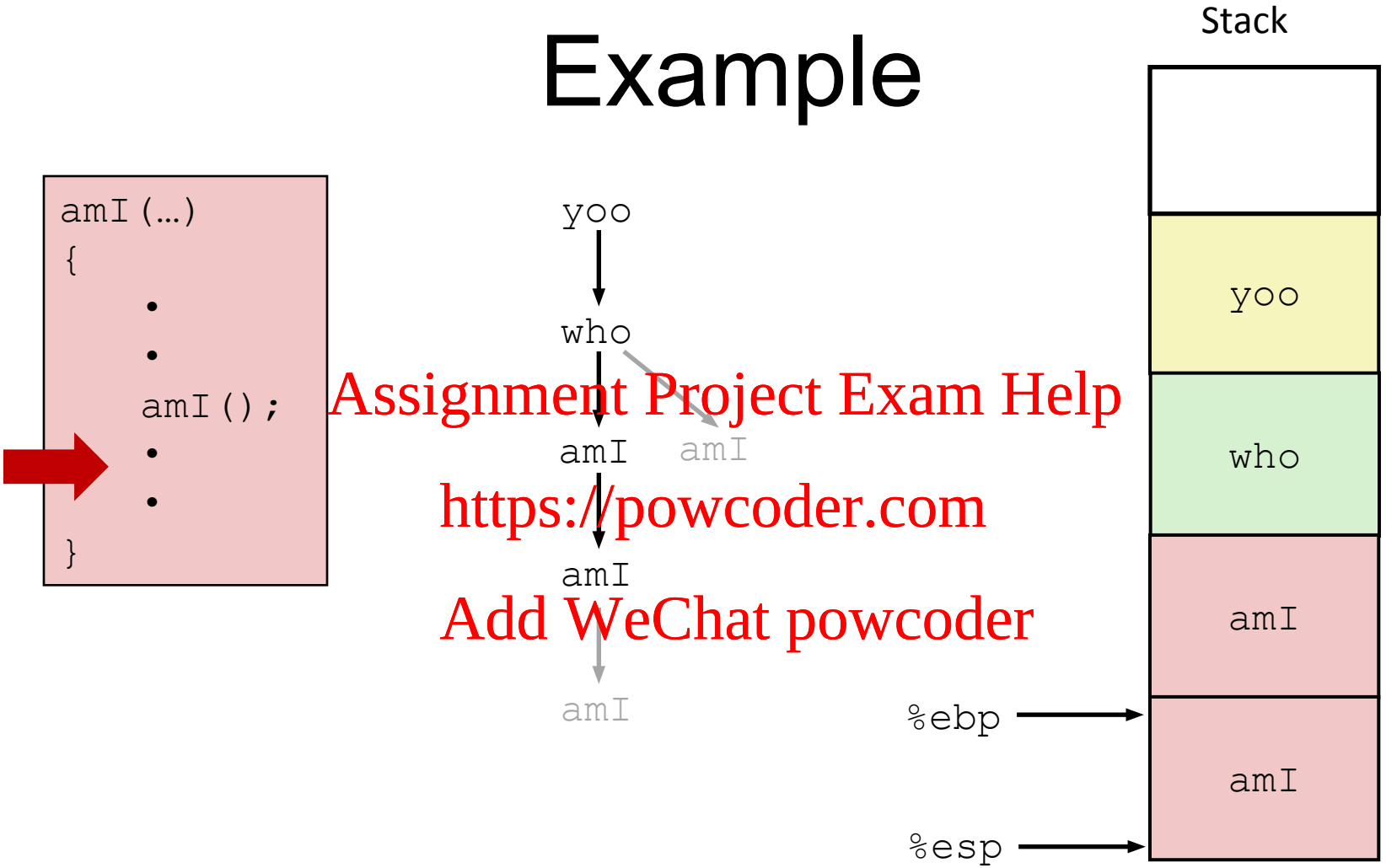
Example



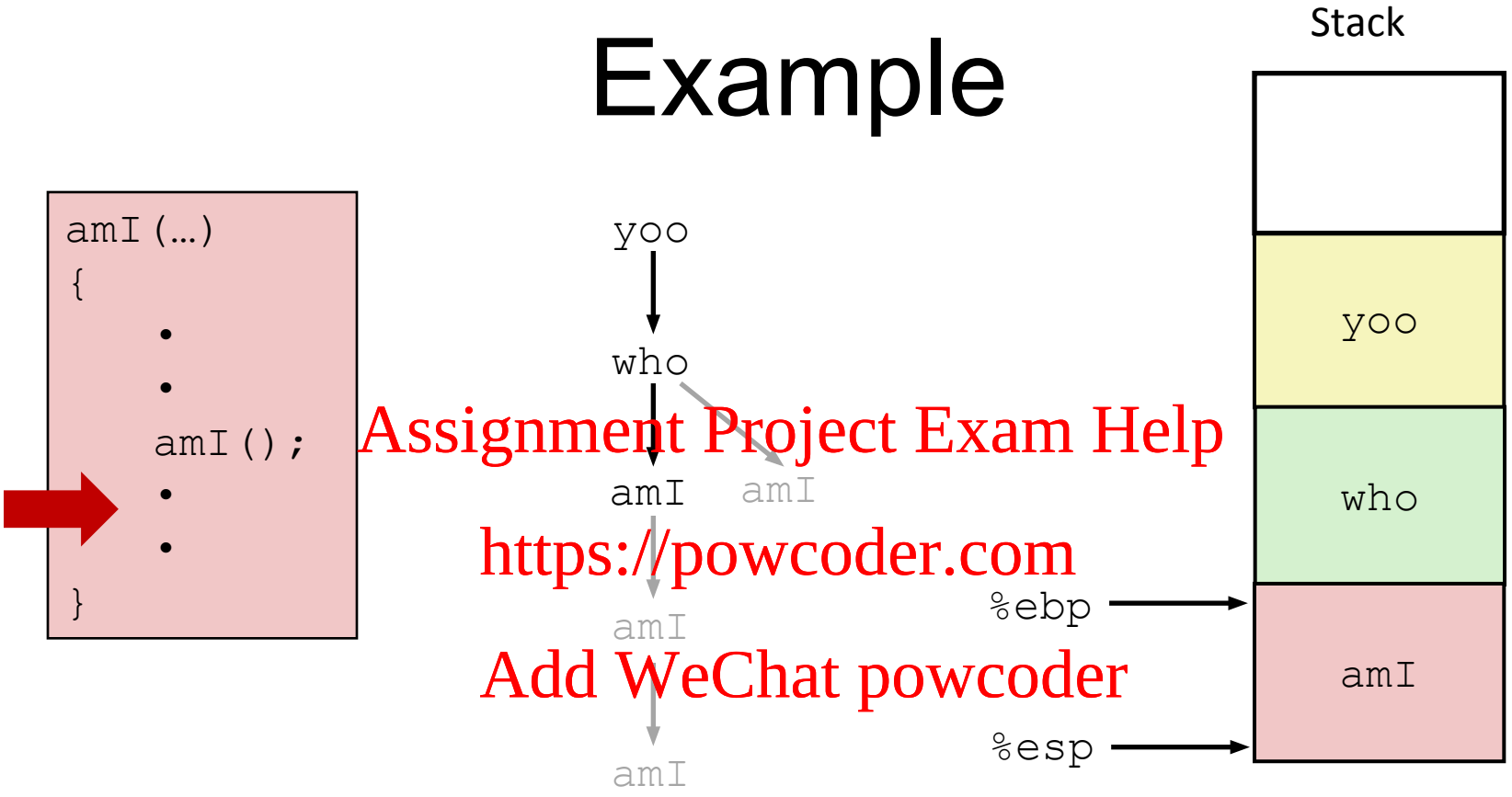
Example



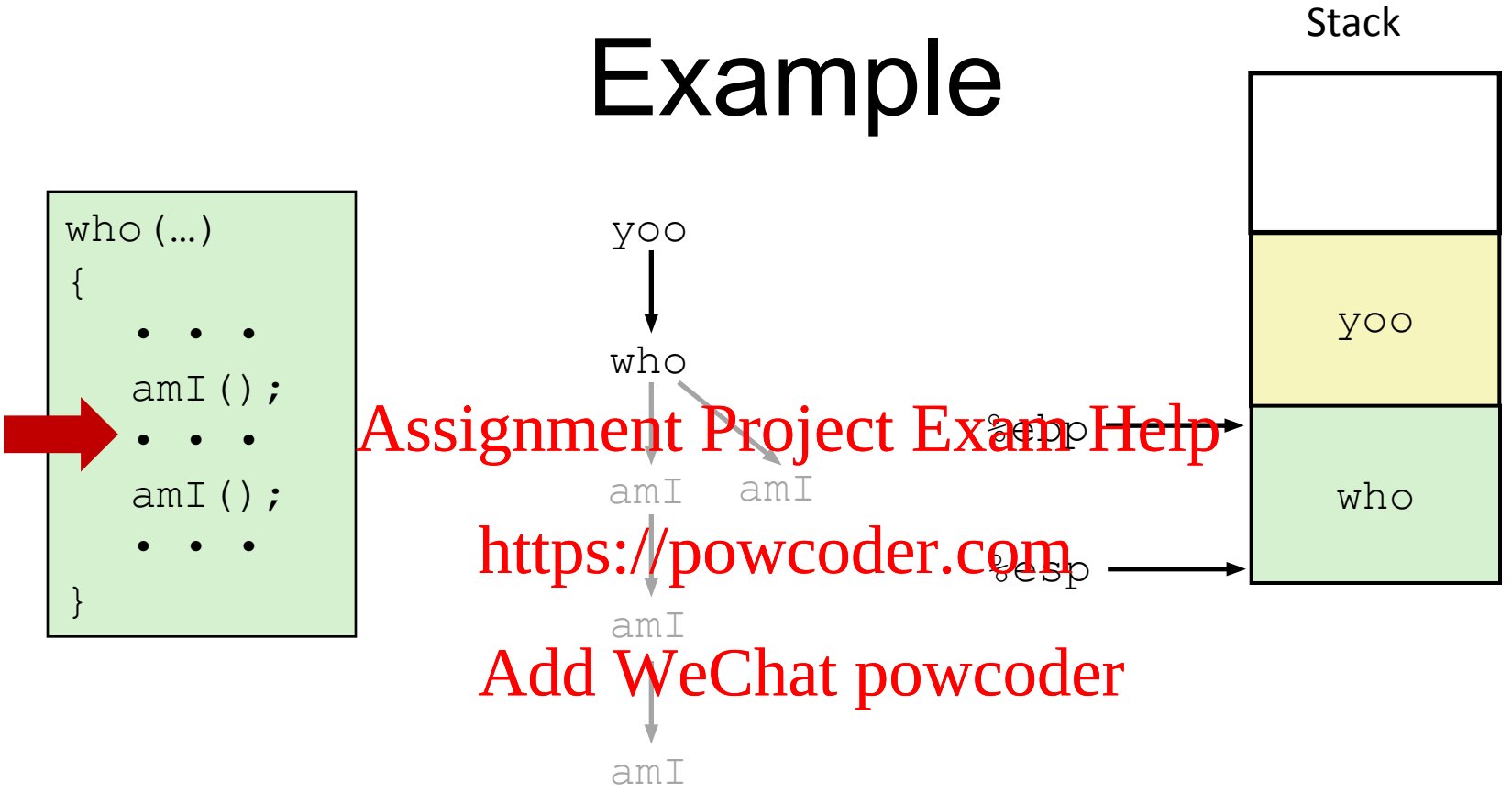
Example



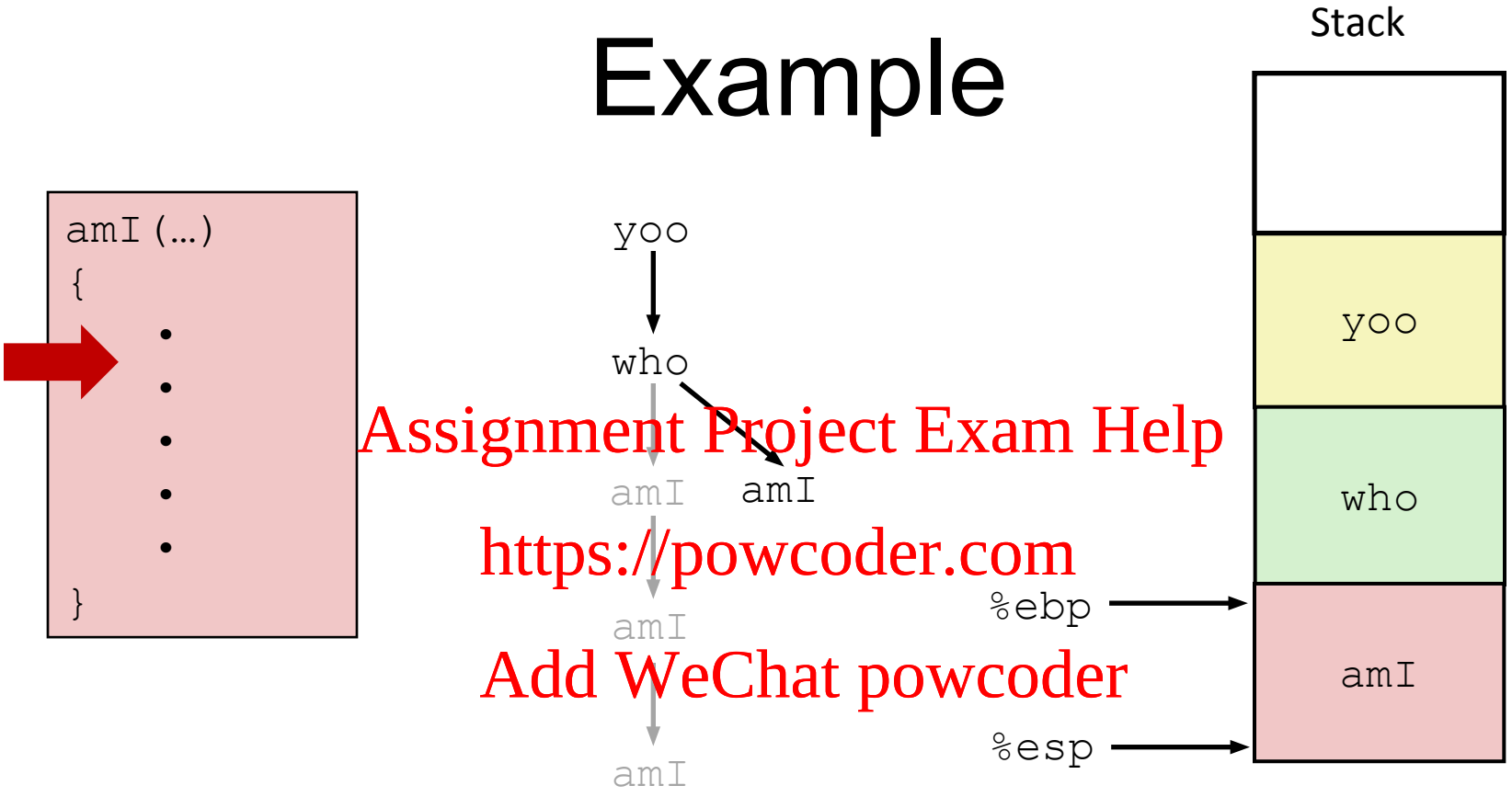
Example



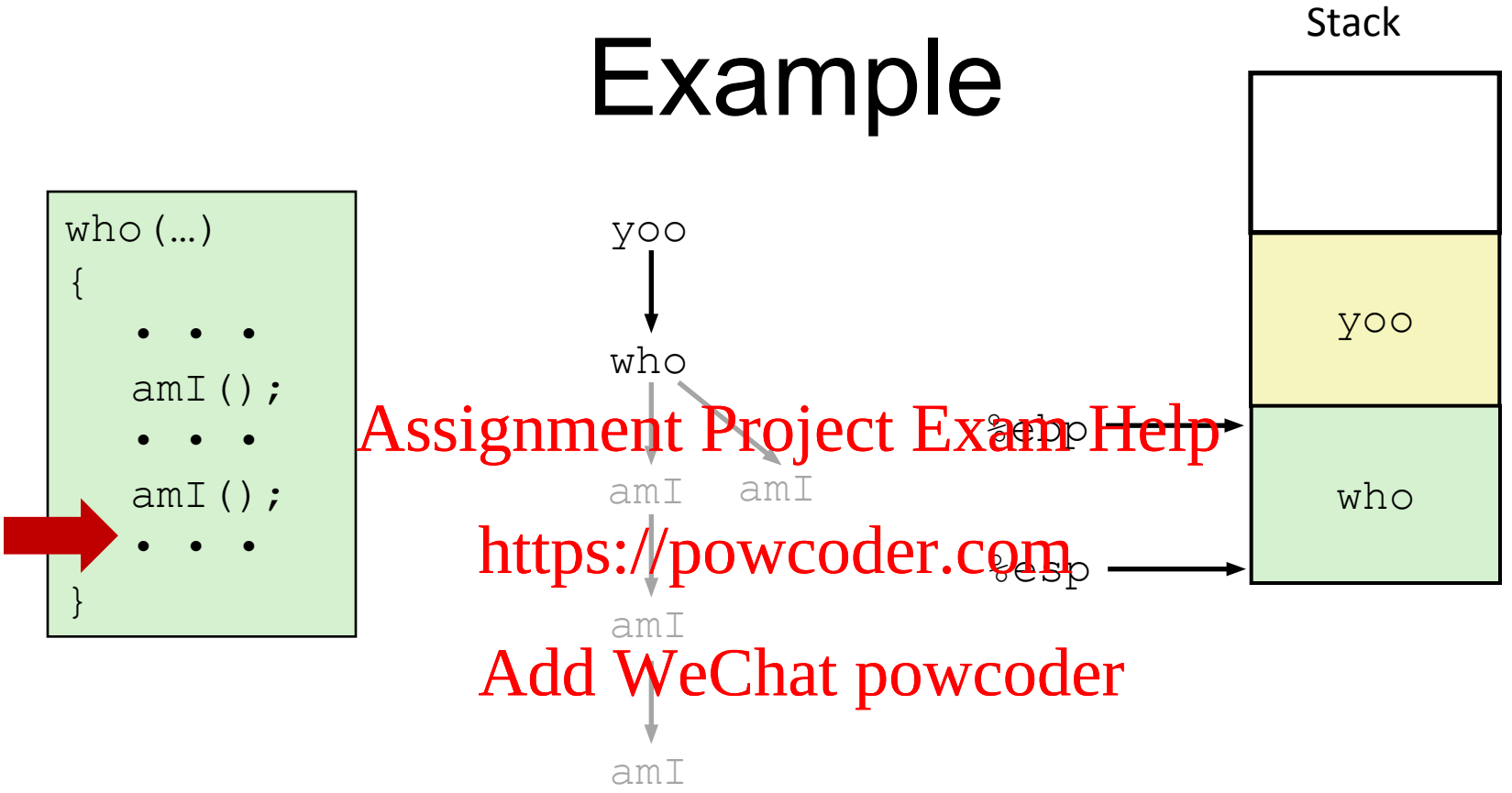
Example



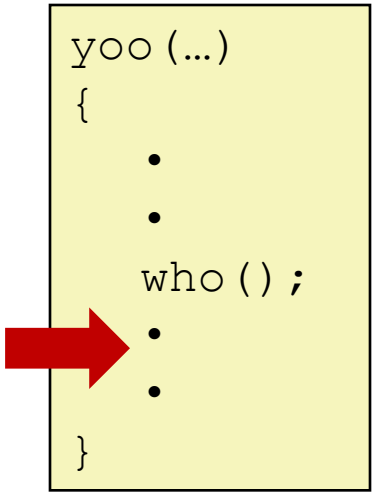
Example

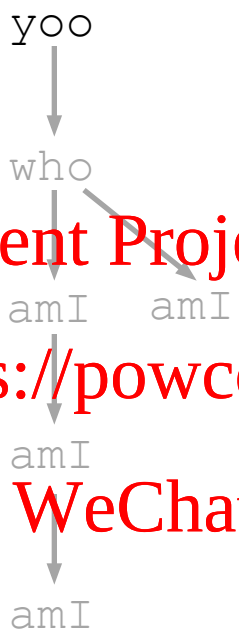


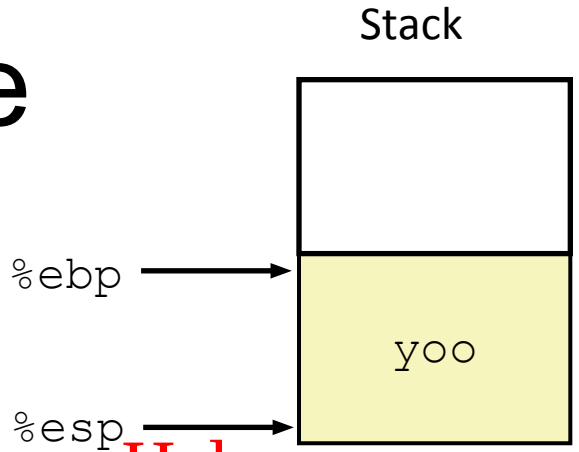
Example



Example

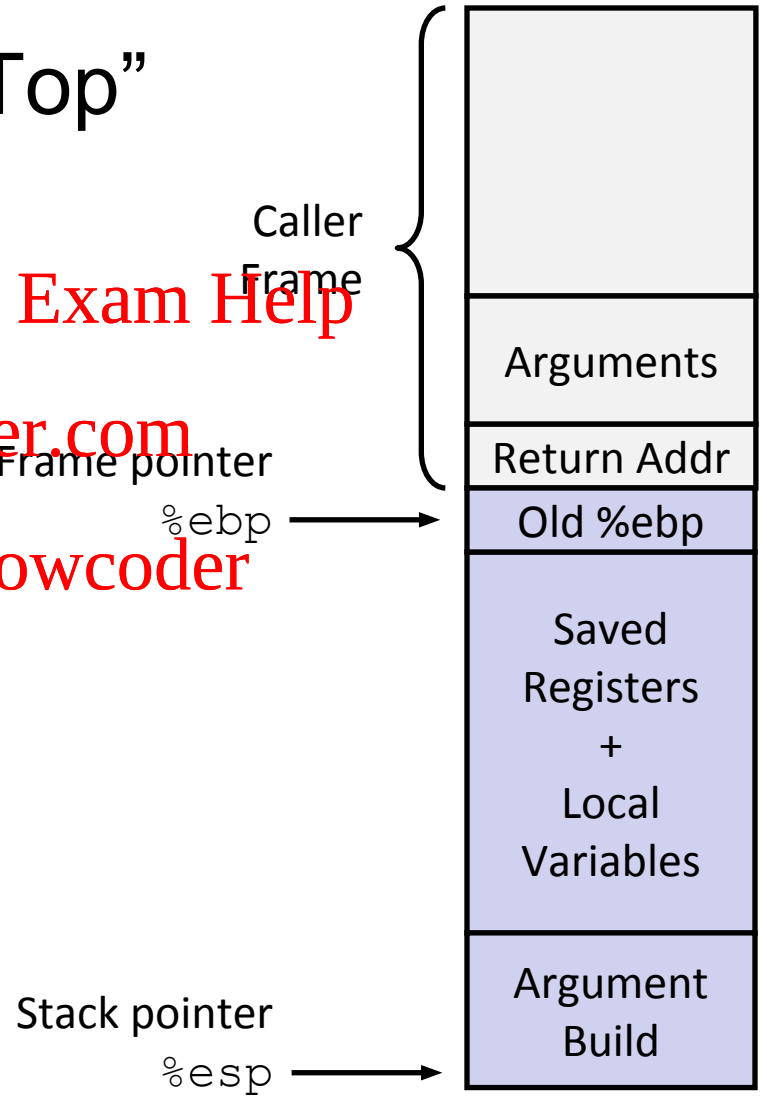






IA32/Linux Stack Frame

- 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`

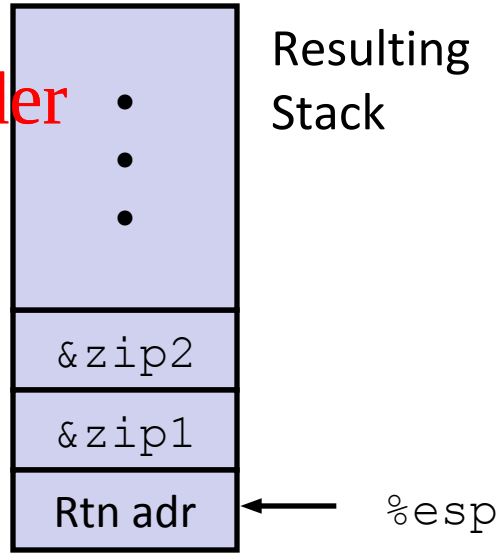
```
int zip1 = 15213;
int zip2 = 91125;

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

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;
}
```

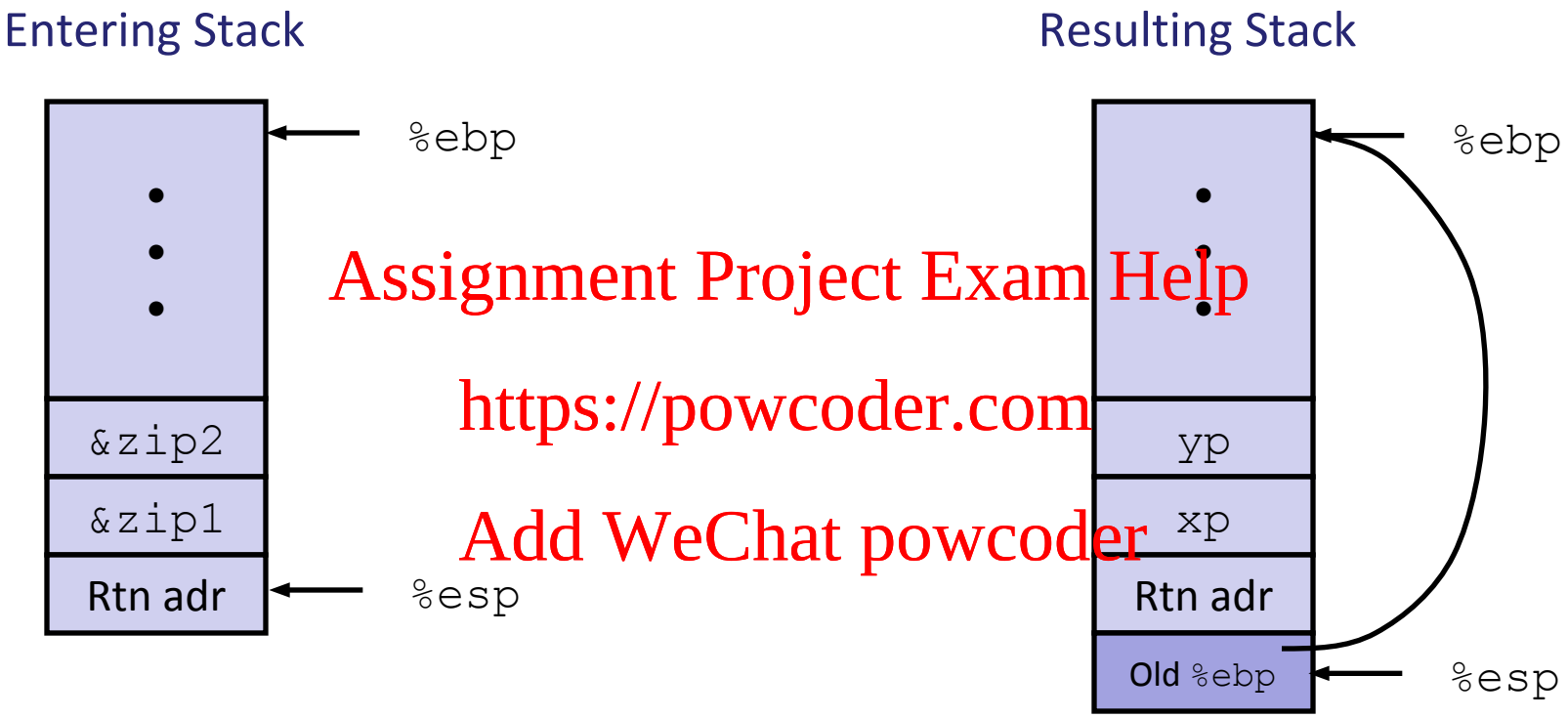
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

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

Set Up

Body

Finish

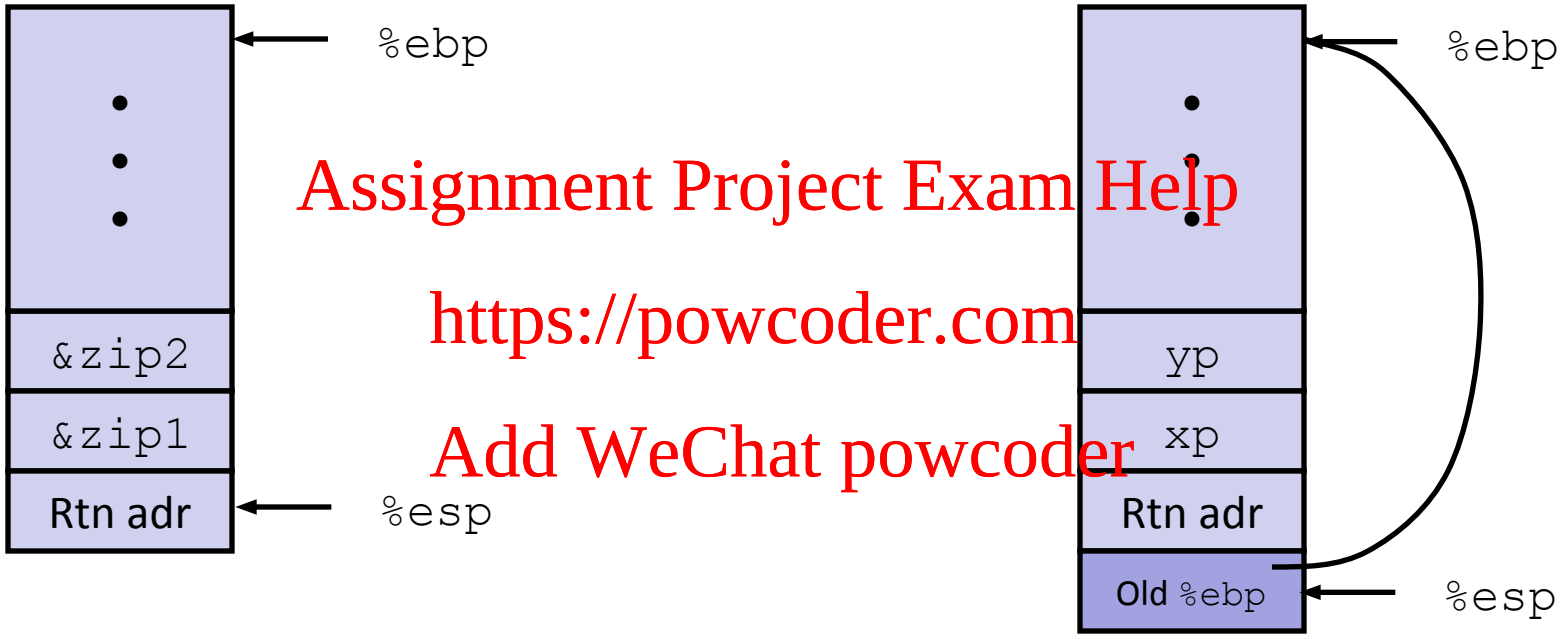
swap Setup #1




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

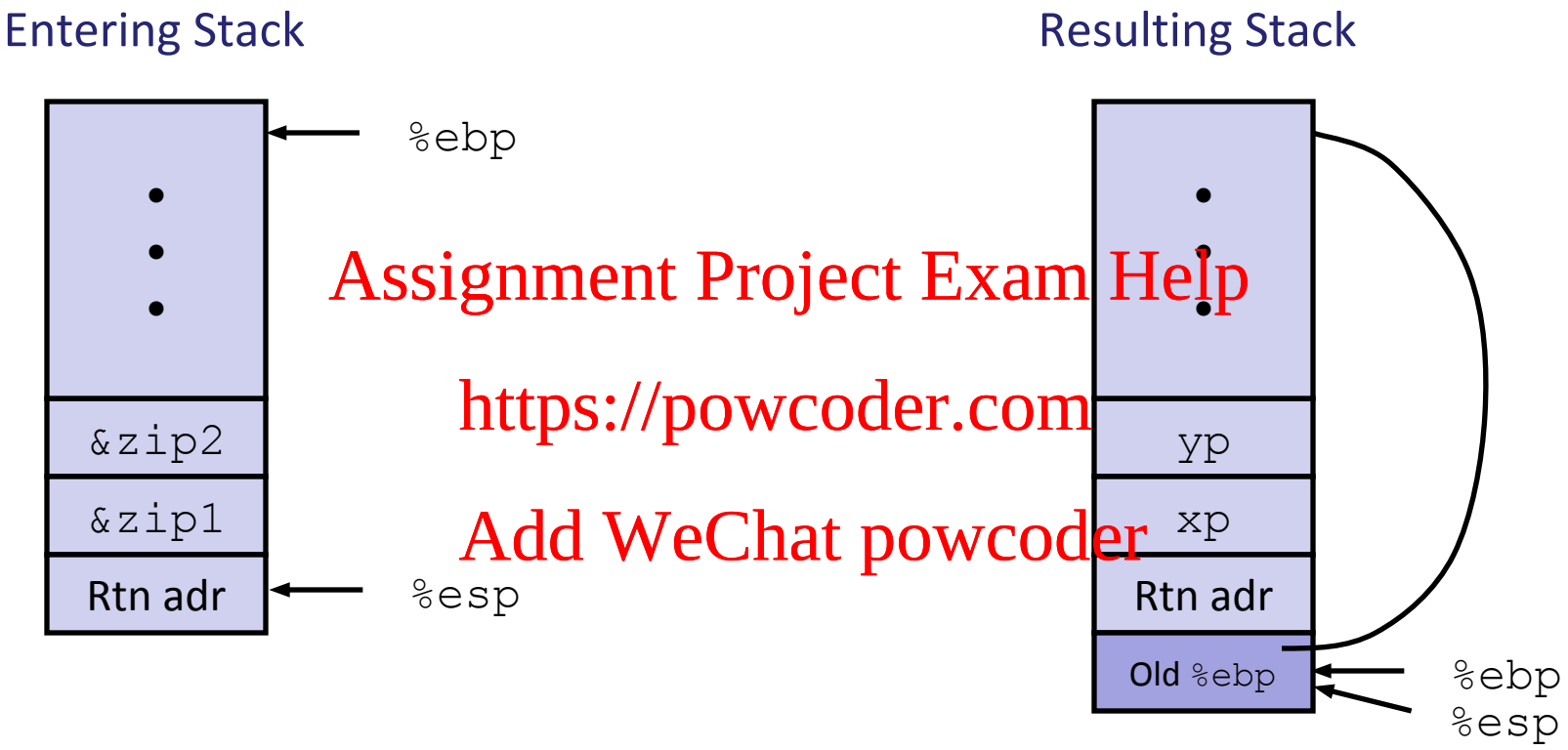
swap Setup #1

Entering Stack




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

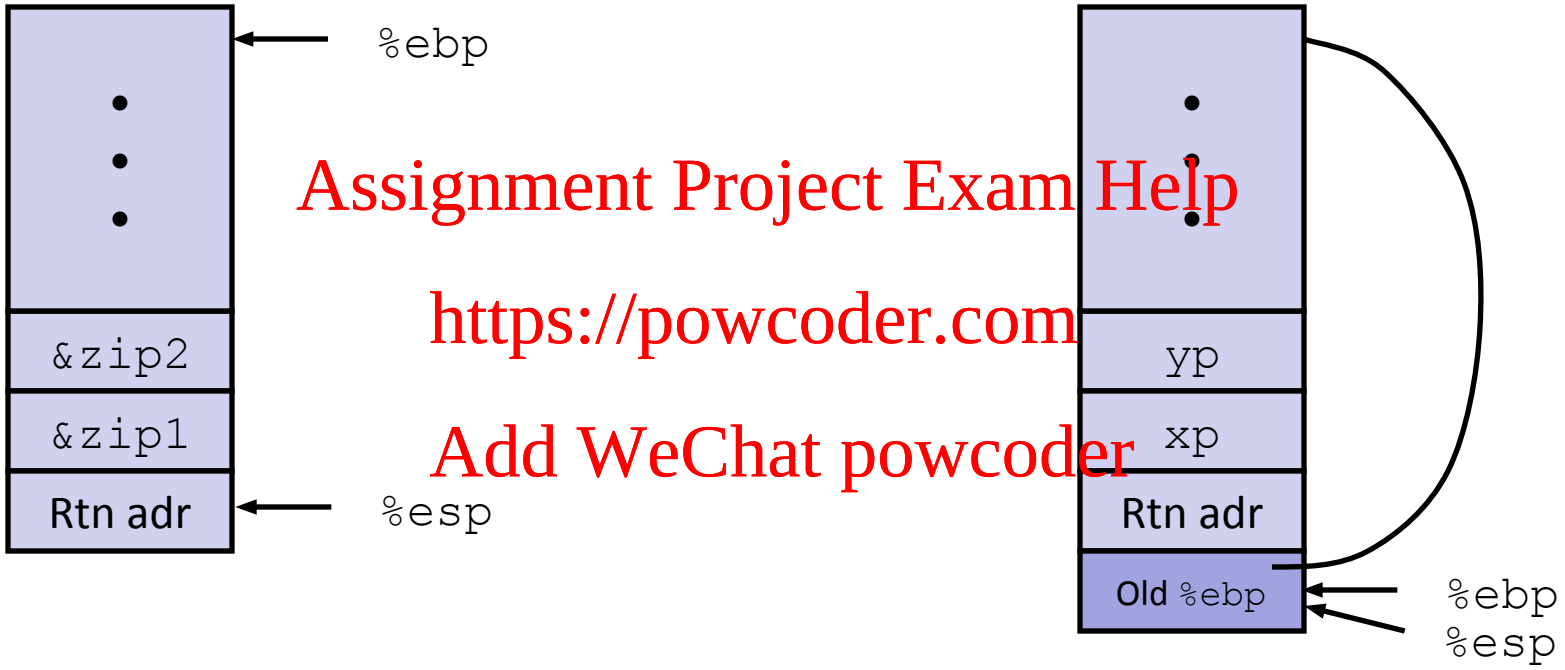
swap Setup #1




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

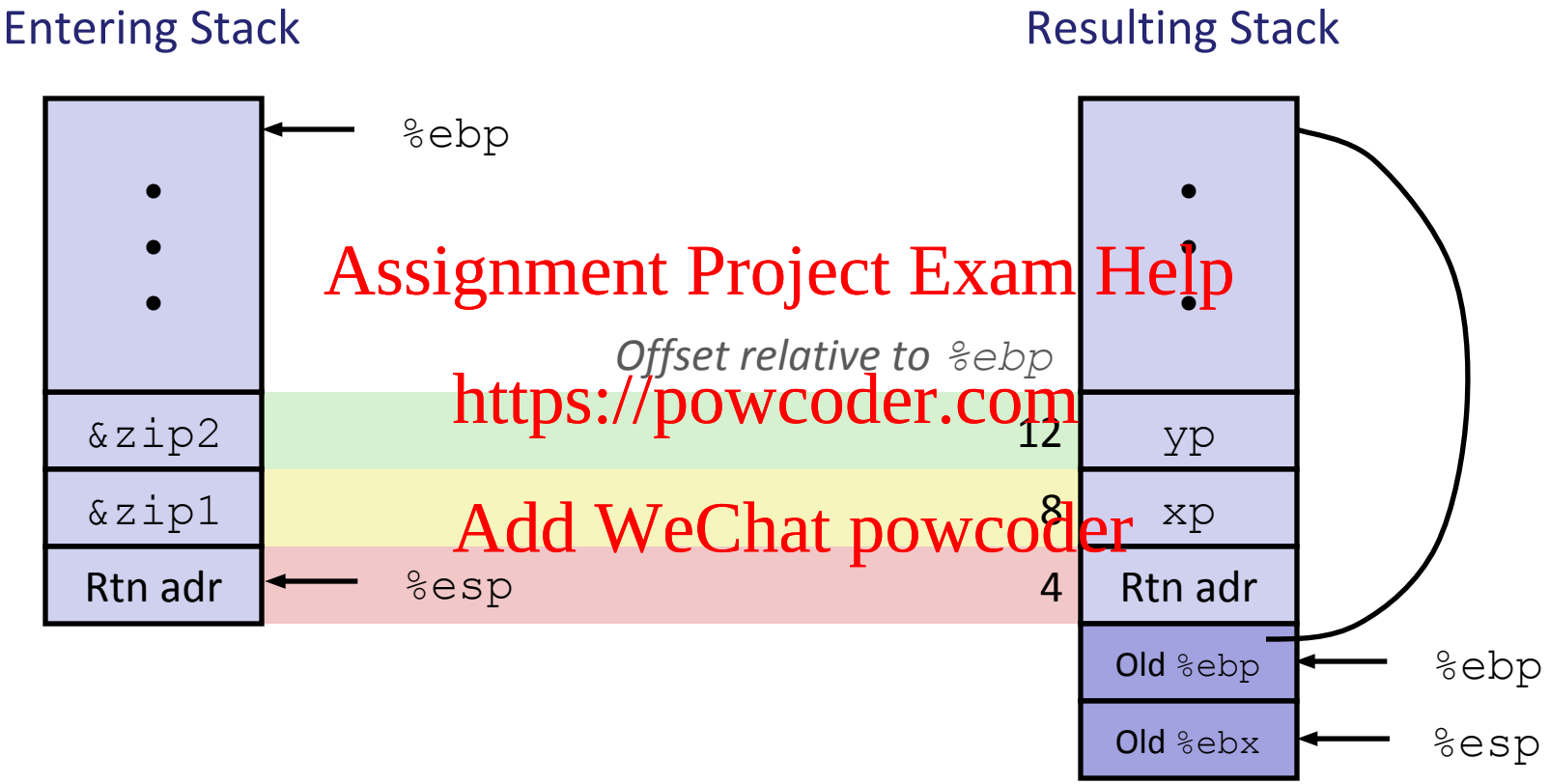
swap Setup #1

Entering Stack




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

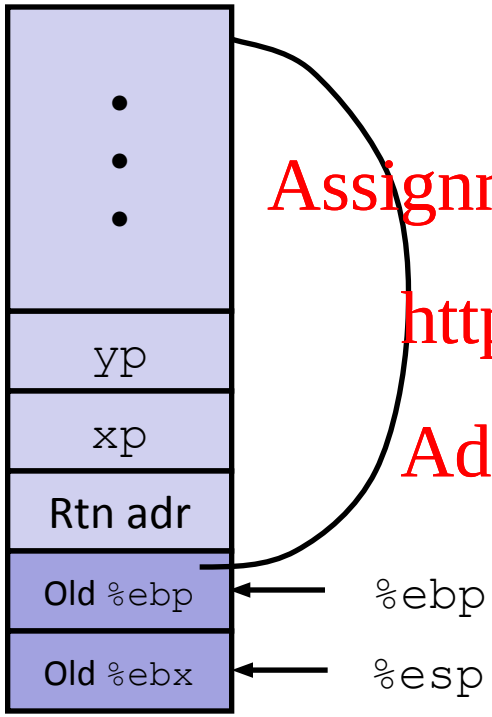
swap Setup #1



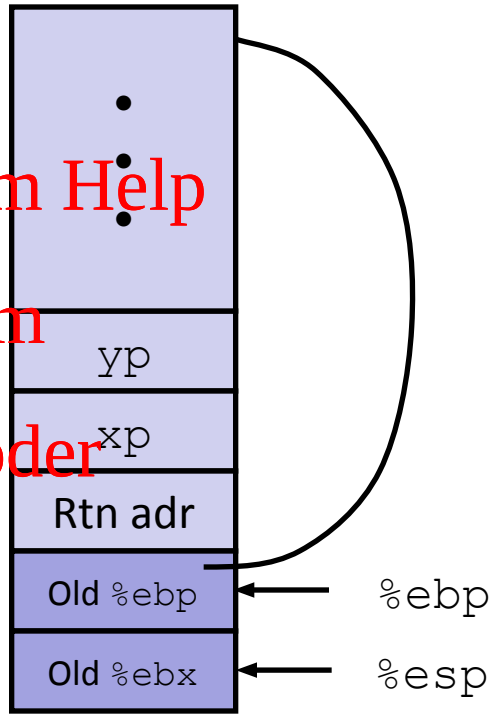

```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

swap Finish #1

swap's Stack



Resulting Stack

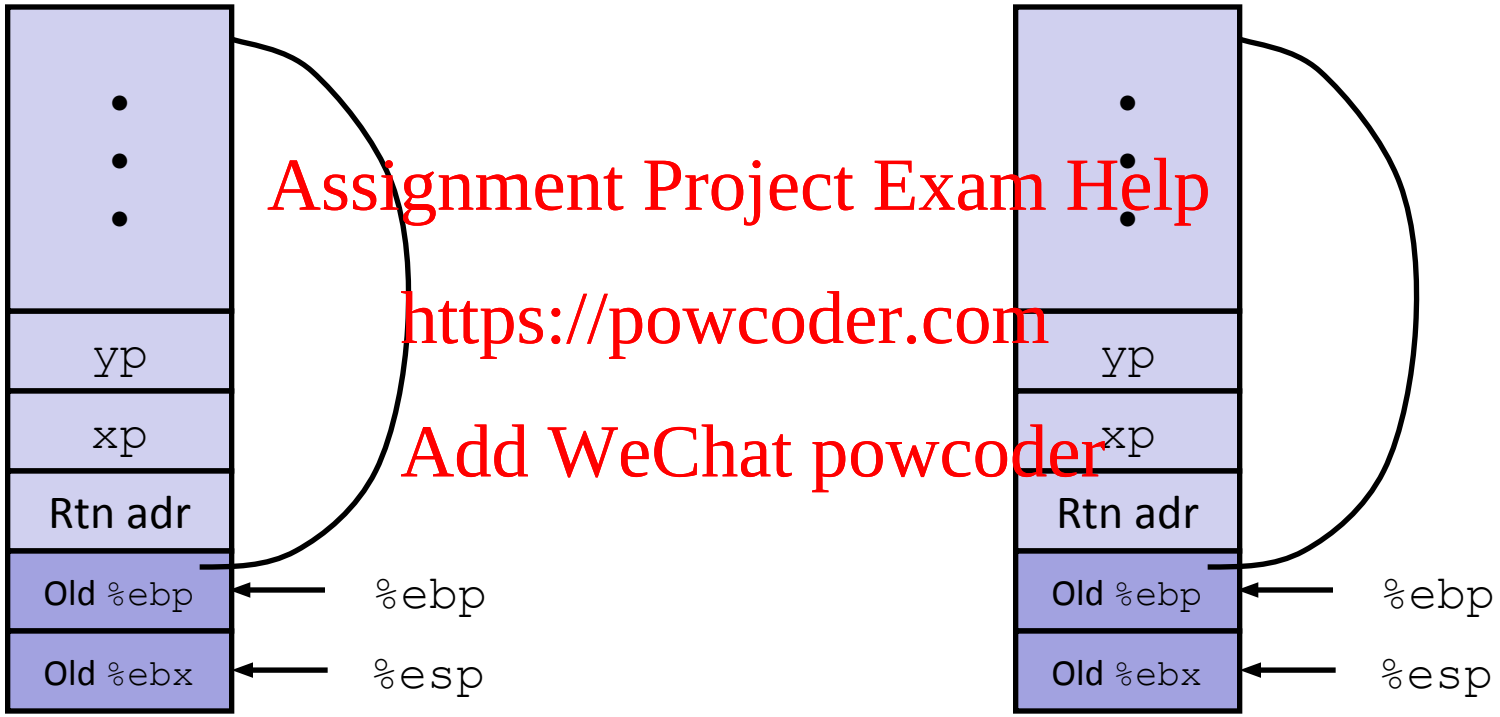



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

swap Finish #2

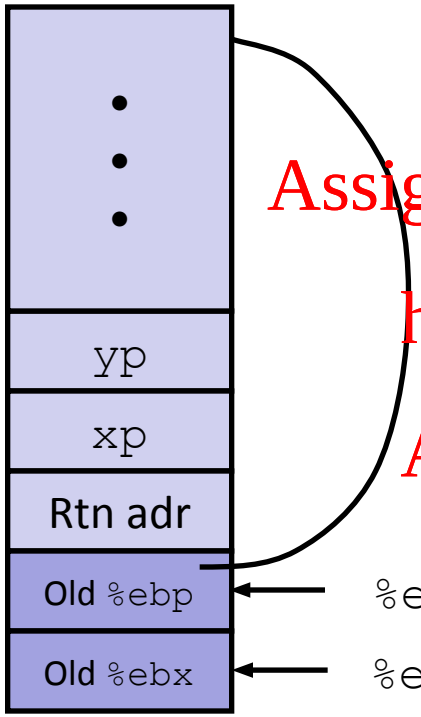
swap's Stack



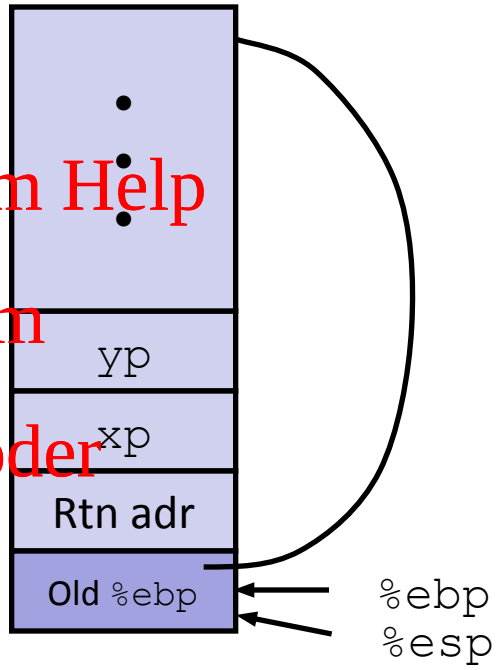

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #2

swap's Stack



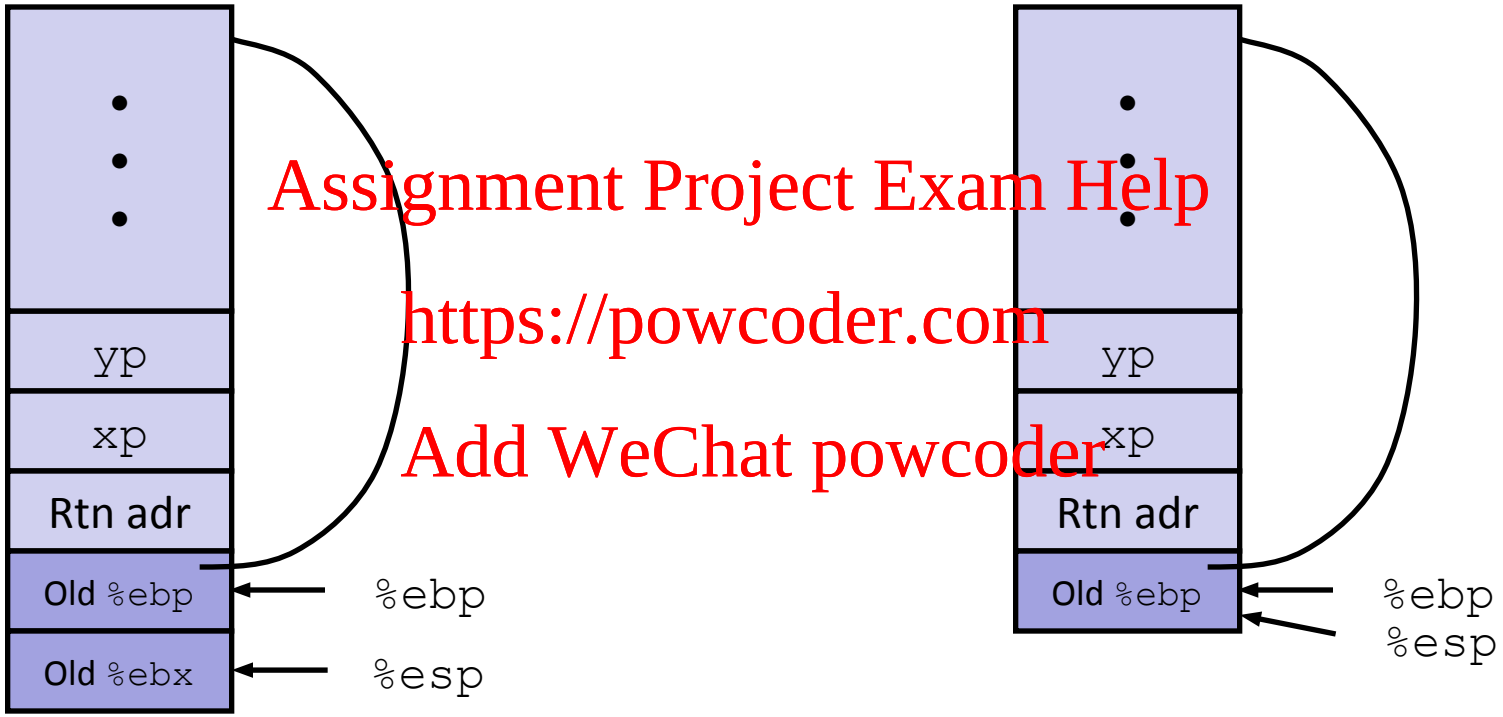
Resulting Stack




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

swap Finish #2

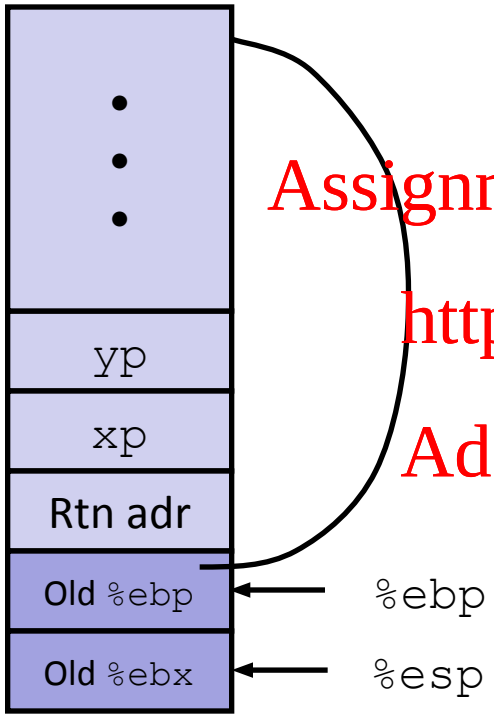
swap's Stack



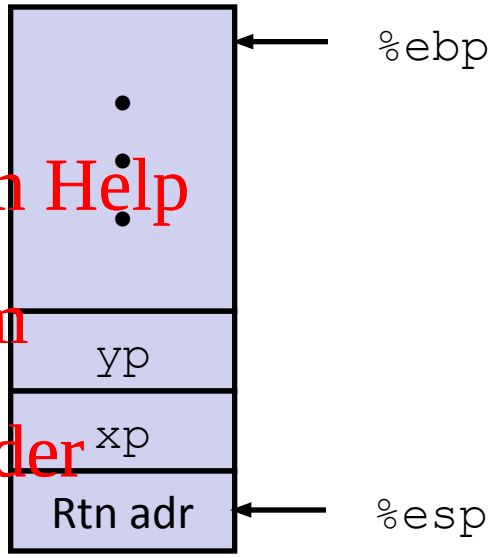

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

swap's Stack



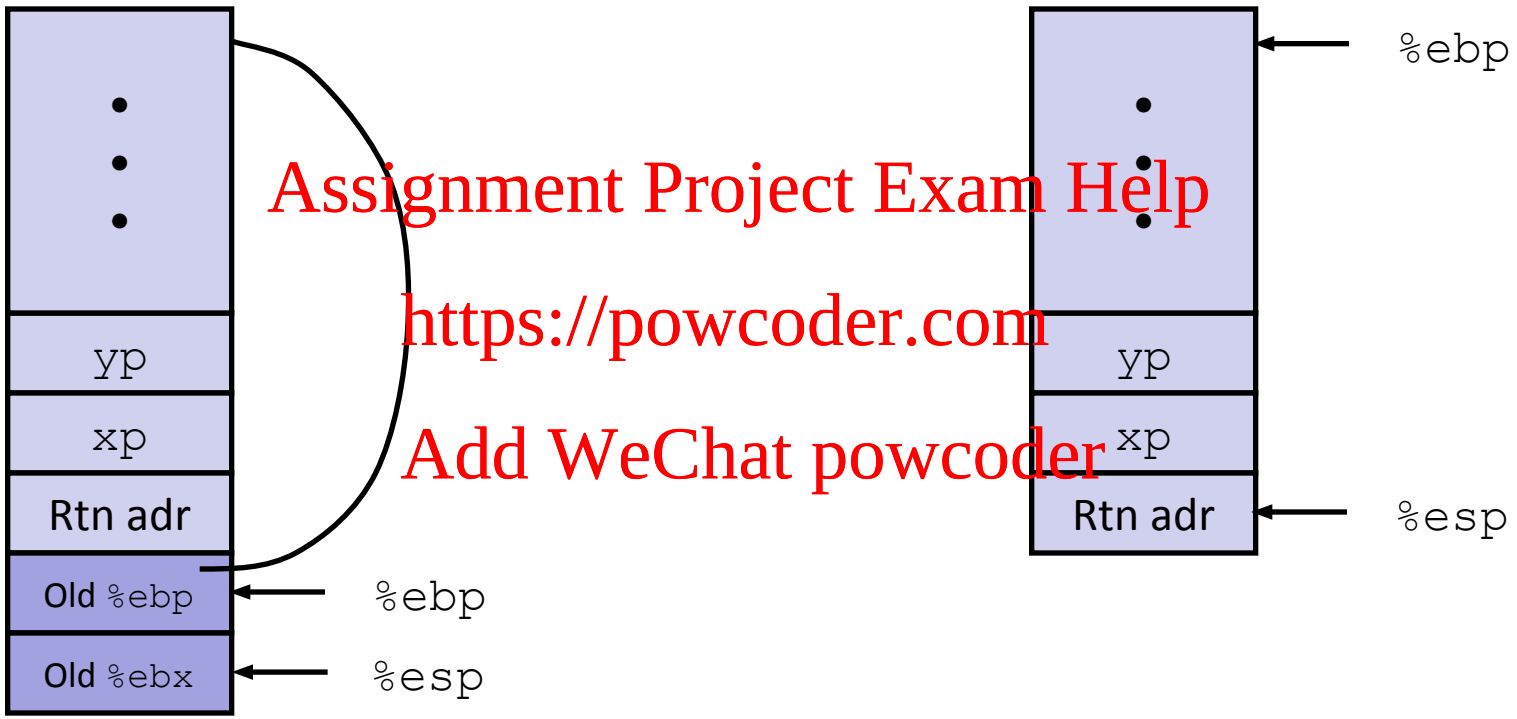
Resulting Stack




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

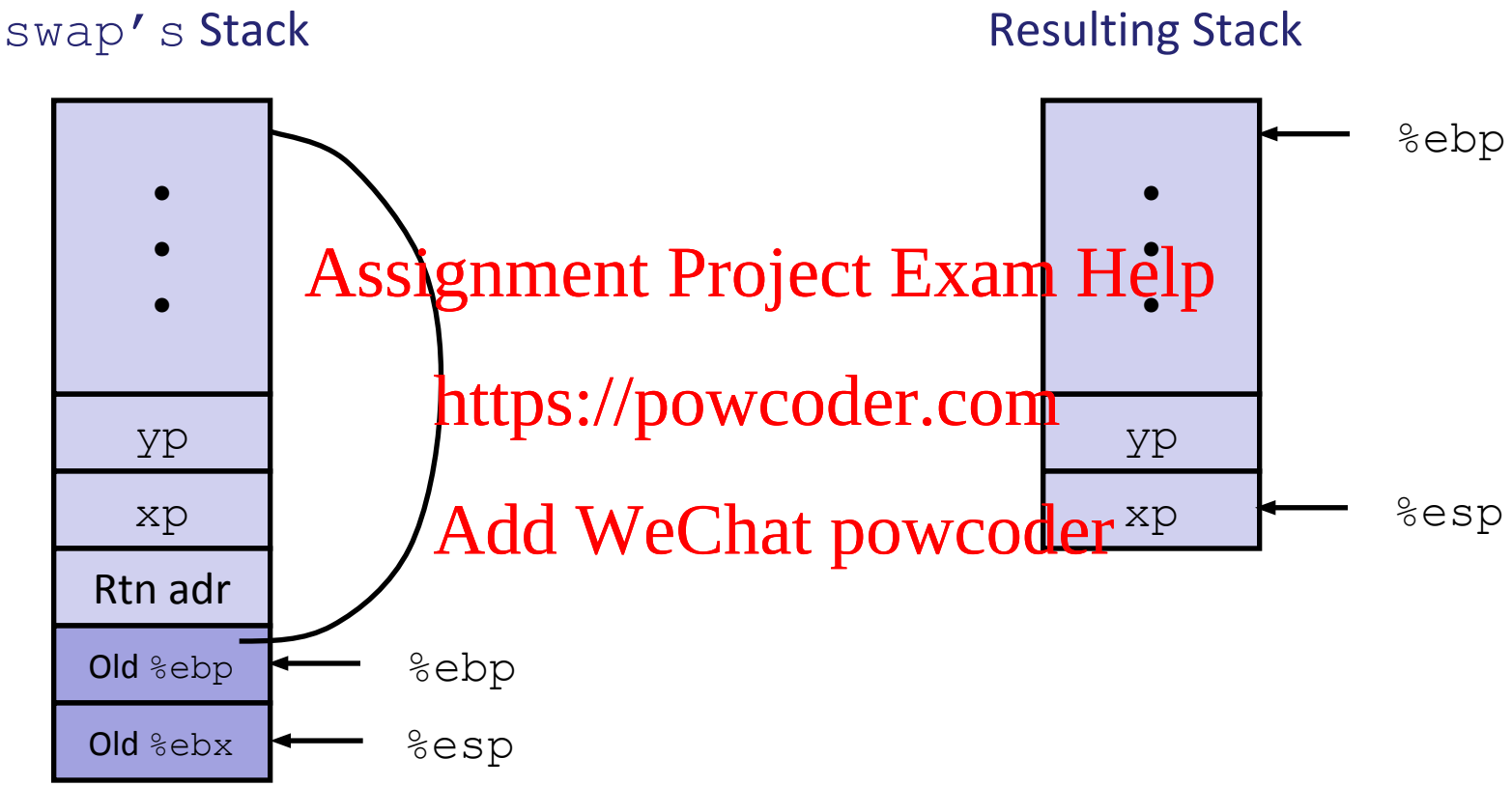
swap Finish #4

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`

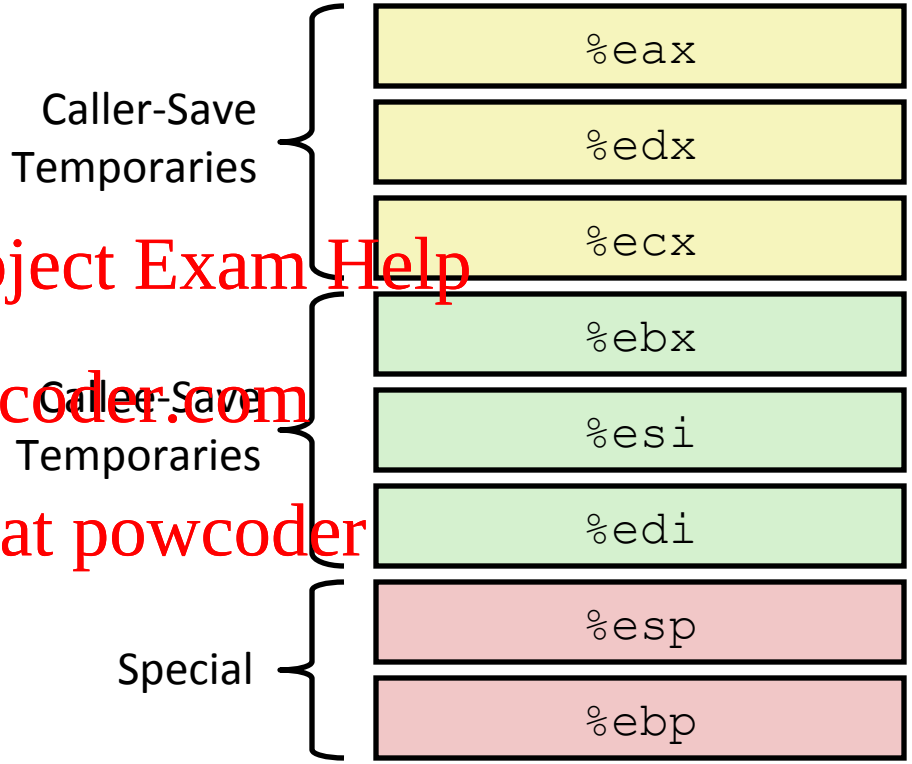
```
080483a4 <swap>:
 80483a4:   55              push   %ebp
 80483a5:   89 e5           mov    %esp,%ebp
 80483a7:   53              push   %ebx
 80483a8:   8b 55 08        mov    0x8(%ebp),%edx
 80483ab:   8b 4d 0c        mov    0xc(%ebp),%ecx
 80483ae:   8b 1a           mov    (%edx),%ebx
 80483b0:   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
```

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



More to Come

- In Module 2, we will revisit
  - Program Structure
  - Binaries
  - Static & Dynamic Libraries
  - Memory/Storage

Assignment

- For Wednesday
  - HTAOE Ch. 2 69-80
- For Monday
  - Complete the Python (version 2.7) tutorial at http://docs.python.org/tutorial/index.html
    - Select version in upper left corner
    - Even proficient Python programmers will learn something