Assembly Review

NMSU Reverse Engineering
Joshua Reynolds
Spring 2024
With slides from CS 461 at UIUC

Intel vs AT&T Syntax

see https://staffwww.fullcoll.edu/aclifton/courses/cs241/syntax.html

AT&T Syntax

Items in () are memory locations

Immediate values have a $ in front of them: $0xEF

Registers have a % like: %eax

// starts a comment

1st operand is source, 2nd is destination

Intel Syntax (used in book)

Items in [] are memory locations

Instructions don’t say operand sizes

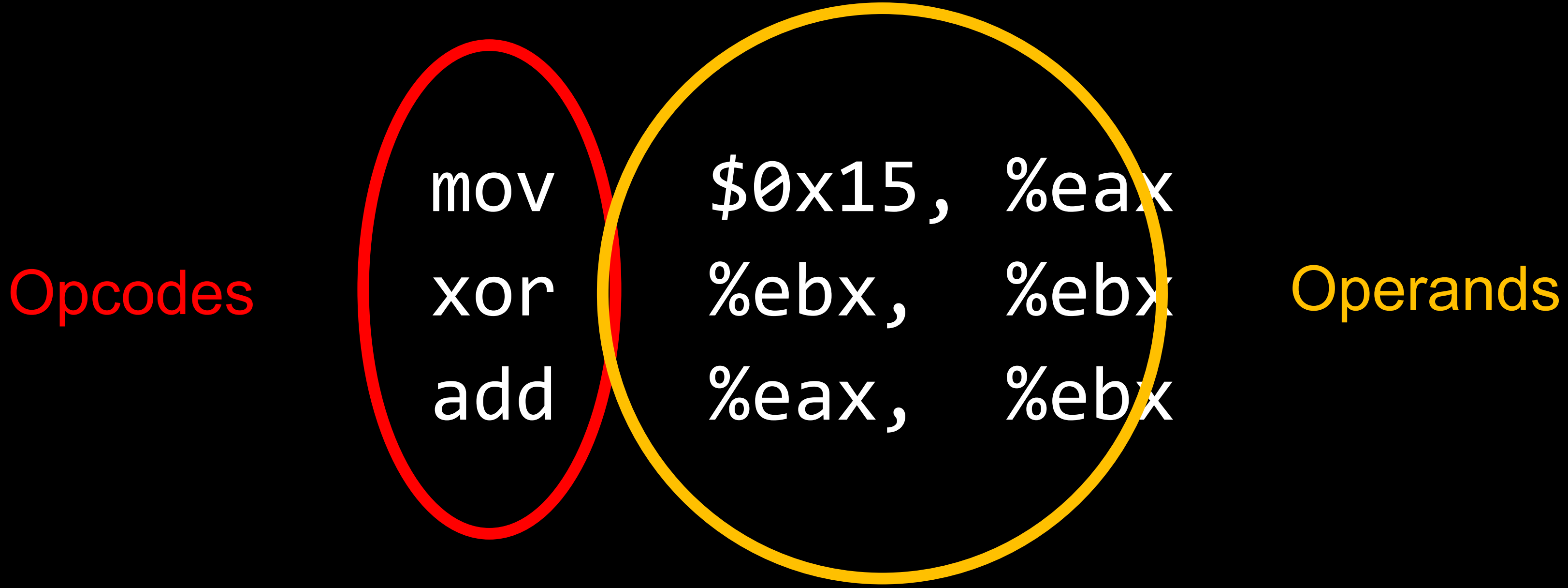
; Is the comment character

1st operand is destination, 2nd is source

Review: x86 Assembly

```
mov    $0x15, %eax
xor    %ebx,  %ebx
add    %eax,  %ebx
```

Review: x86 Assembly



Opcodes

```
mov   $0x15, %eax
xor   %ebx,  %ebx
add   %eax,  %ebx
```

Operands

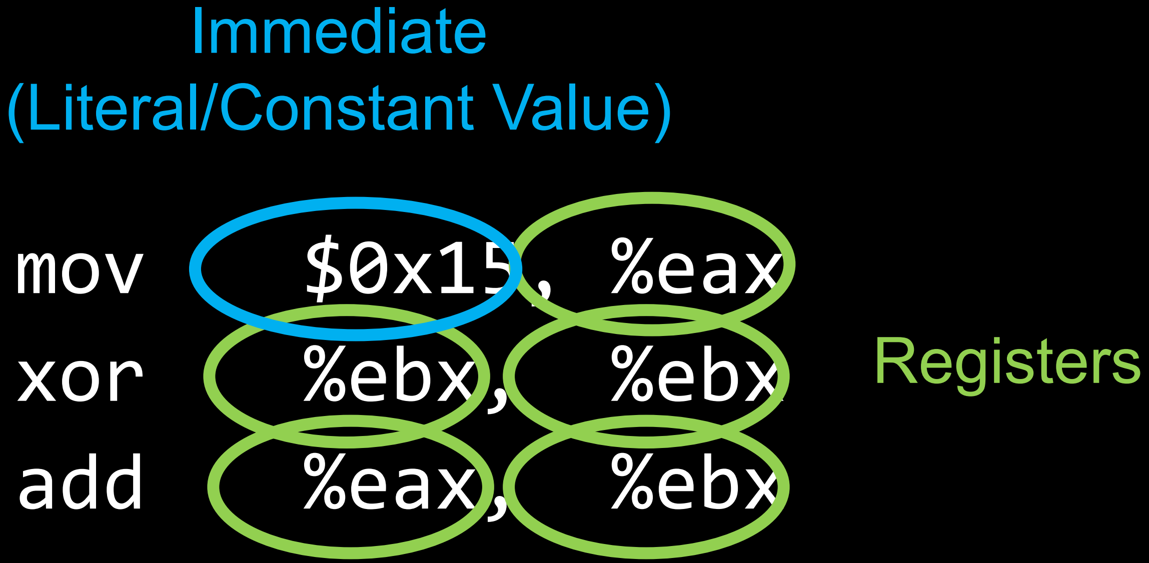
Review: x86 Assembly

Immediate
(Literal/Constant Value)

```
mov   $0x15, %eax
xor   %ebx,  %ebx
add   %eax,  %ebx
```

Registers

Review: x86 Assembly



Immediate
(Literal/Constant Value)

```
mov    $0x15, %eax
xor    %ebx, %ebx
add    %eax, %ebx
```

Registers

Also, memory addresses (more on these in a moment)

Commonly Used x86 Registers

General purpose registers

- EAX - Return value
- EBX
- ECX - Loop counter
- EDX
- EDI - Repeated destination
- ESI - Repeated source

Special Registers

- EBP – Frame pointer/Base pointer
- ESP - Stack pointer
- EIP - Program counter
- EFLAGS - Status of previous operations (used in conditionals)

x86 Assembly Syntax

There are two main variants of x86 syntax:

Intel

- `add eax, [ebx+4]`
- Destination operand first, then source
- Brackets indicate memory access

AT&T (GAS)

- `add 4(%ebx), %eax`
- Source operand first, then destination
- Parentheses indicate memory access

In this week's assignment, the assembler expects AT&T syntax

Memory Operations

- What if we want to use a value from memory, rather than a register or constant value?

Example: `Load Mem[%ebp + 8 + (4 * %ecx)] into %eax`

- x86 Assembly provides a specific syntax for accessing memory locations

```
mov 8(%ebp,%ecx,4), %eax
```

AT&T Memory Address Calculation

Write it:

```
displacement (base_reg, offset_reg, multiplier)
```

Calculate it:

```
base_reg + displacement + (offset_reg*multiplier)
```

```
mov    8 (%ebp), %eax          # Mem[EBP+8] to eax
mov    12 (,%edx,4), %eax      # Mem[EDX*4+12] to eax
```

Notice that not all fields are required!

GAS/AT&T Memory Syntax Example

```
typedef struct {
  int a, b, c, d;
} foo_t;
foo_t my_foos[10];

my_foos[5].c = 479;
```

GAS/AT&T Memory Syntax Example

```
typedef struct {
  int a, b, c, d;
} foo_t;
foo_t my_foos[10];

my_foos[5].c = 479;
```

```
Assume %ebx points to my_foos
mov $5, %ecx
movl $461, 8(%ebx, %ecx, 16)
```

Common x86 Instructions (Opcodes) (1)

Arithmetic Operations

- `add`, `sub` - add/subtract data in first operand to/from second
- `inc`, `dec` - increment/decrement operand
- `neg` - change sign of operand

Logical Operations

- `and`, `or`, `xor` - bitwise and/or/xor
- `not` - flip all of the bit values
- `shl`, `shr` - shift bits left/right

Common x86 Instructions (Opcodes) (2)

Transfer Instructions

- `mov` - copy data from first operand to second
- `lea` - compute address and store it in second operand (does NOT access memory)
- `push` - Push the operand onto the stack (see later slides)
- `pop` - Pop a value off the top of the stack into the operand

Common x86 Instructions (Opcodes) (3)

Transfer Instructions

- `jmp` – jump to label or address specified by operand
- `je` - jump if equal
- `jne` - jump if not equal
- `jz` - jump if zero
- `jg` - jump if greater than
- `jl` - jump if less than
- `jle/jge` - jump if equal or less than/greater than

For conditional jumps, EFLAGS is used. EFLAGS is a register set by the CMP and TEST instructions (and all other arithmetic instructions)

32-bit x86 ISA

- 1 byte = 8 bits
- char -> 1 byte
- integer -> 4 bytes
- word -> 2 bytes (in gdb, word -> 4 bytes)
- Memory address -> 4 bytes
- Pointer -> 4 bytes
- Registers -> 4 bytes
- Each memory location -> 1 byte

How to make a linux syscall in x86 64-bit

Syscall number goes into: RAX
Arg 1 in RDI
Arg 2 in RSI
Arg 3 in RDX
Arg 4 in R10
Arg 5 in R8
Arg 6 in R9

The result (which may be a pointer) comes back in RAX
RCX and R11 may be “clobbered”

The exec() system call

See execve(2) in your man pages

The execve syscall is 59

See:
https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_64.tbl#L11

Activity 1 Intro to GDB

Write a simple hello world program

Compile it with debugging flags (gcc -g)

Run it in the debugger

Then try the code from this week’s assignment

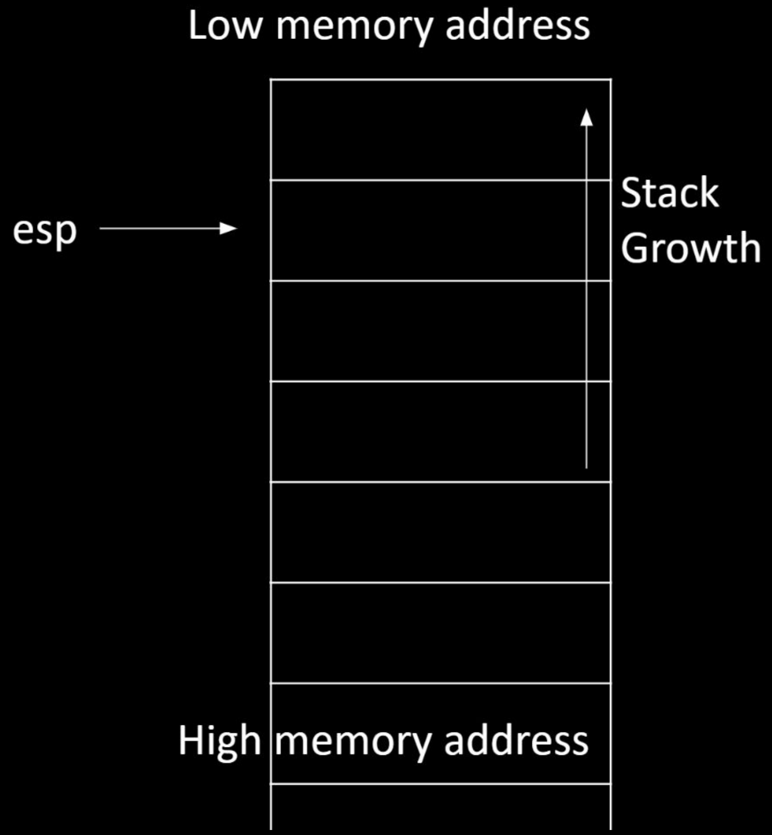
https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

The Stack

- Stores working data (local variables, function arguments, return addresses, etc)
- Last-in First-out (LIFO) structure
- Grows downwards (towards lower memory addresses)
- Manipulated with `push` and `pop` instructions

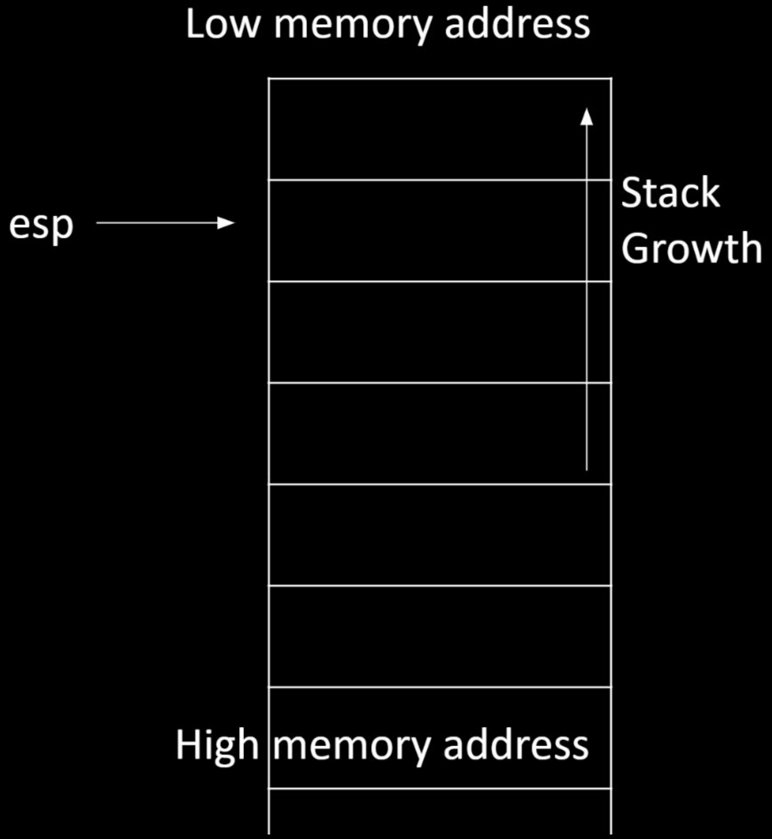
The Stack

- ESP (stack pointer) points to the top of the stack



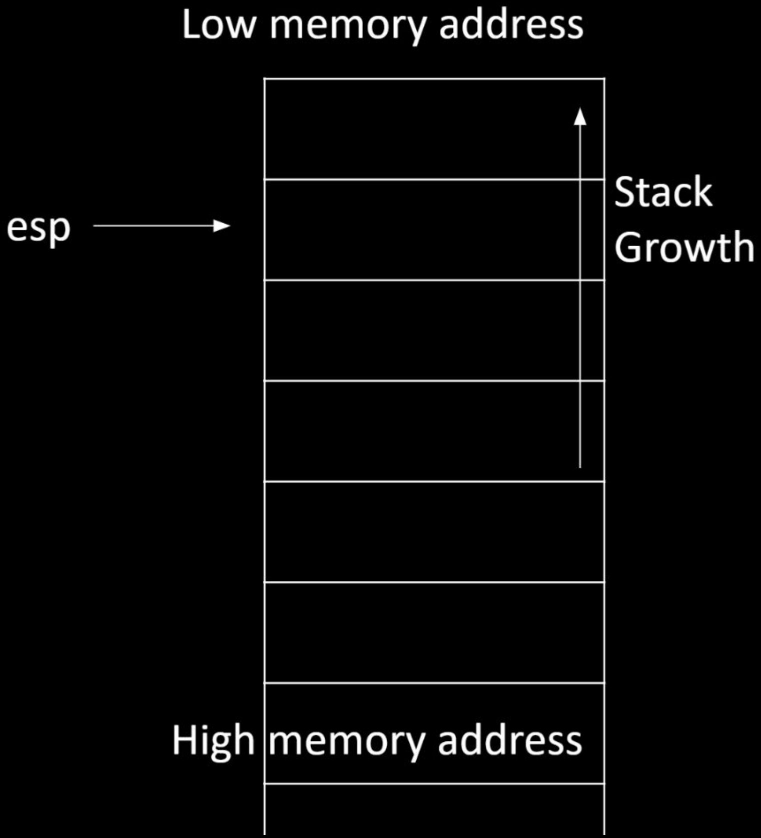
The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction subtracts from ESP and then writes to the top of the stack



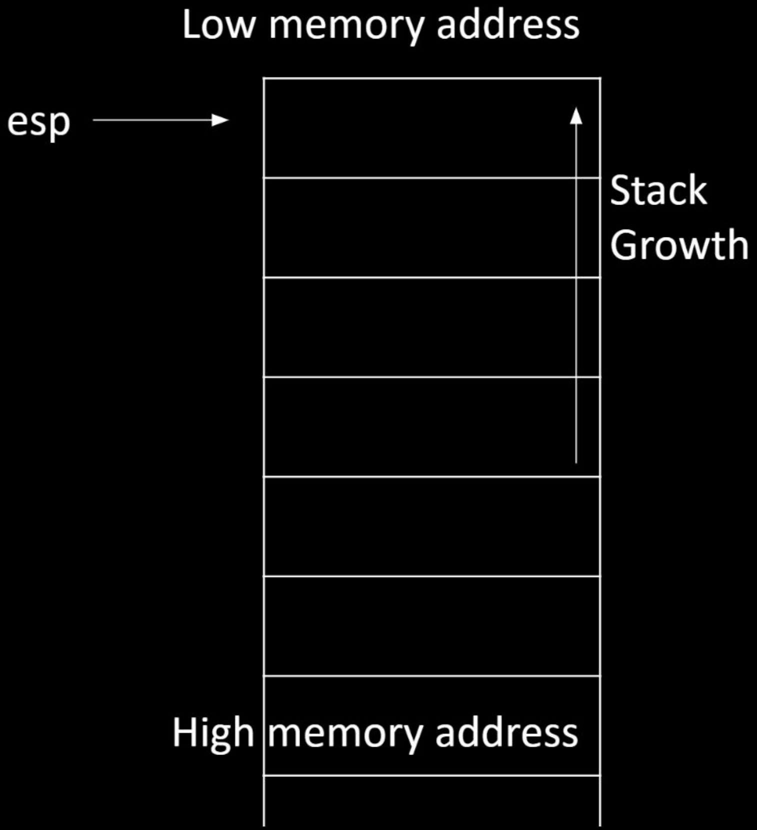
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`



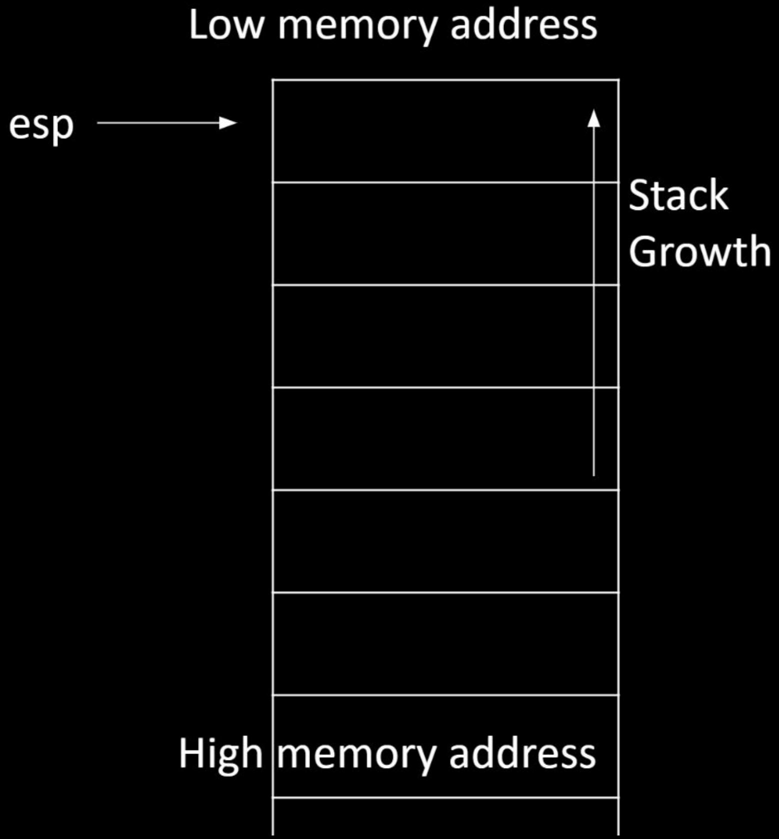
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts from ESP and then writes to the top of the stack
    - Example: `push 0x40404040`



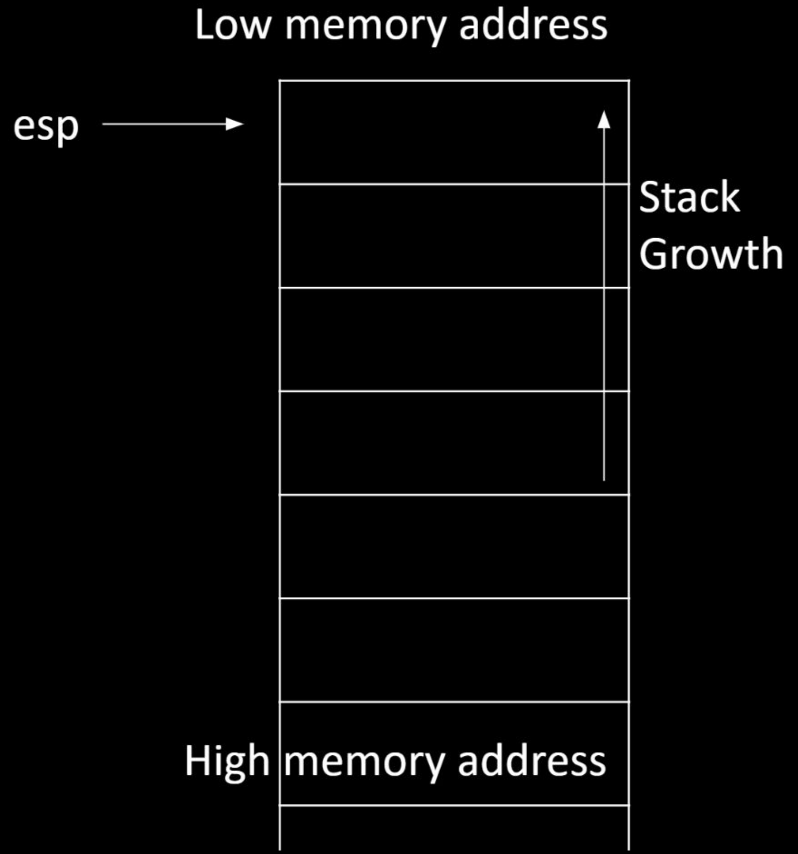
The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction subtracts from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`



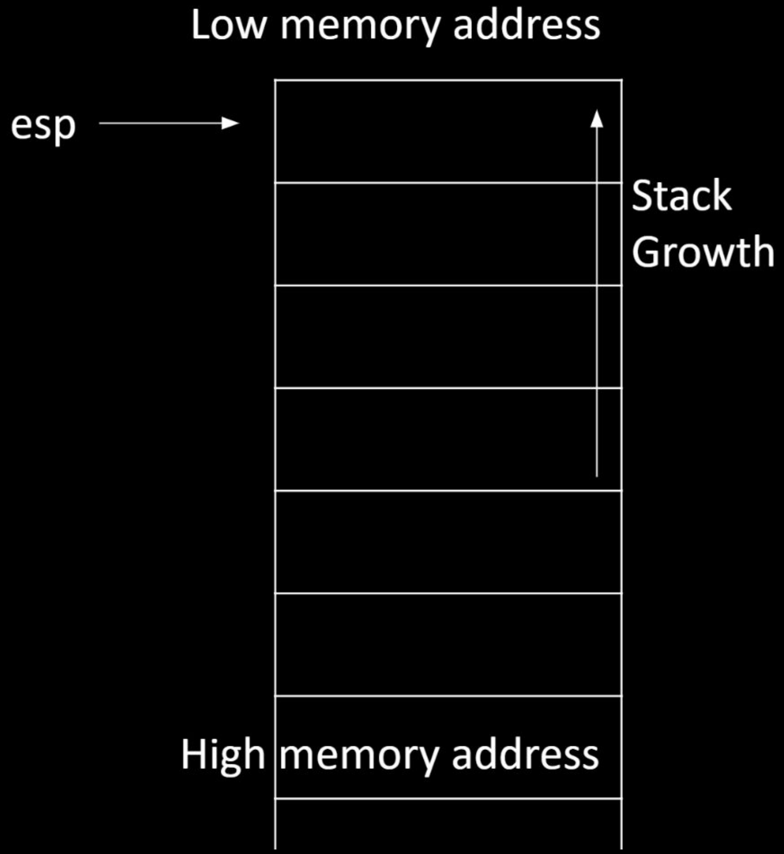
The Stack

- ESP (stack pointer) points to the top of the stack
- `push` instruction subtracts from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`
- pop instruction reads the value on top of the stack and then adds to ESP



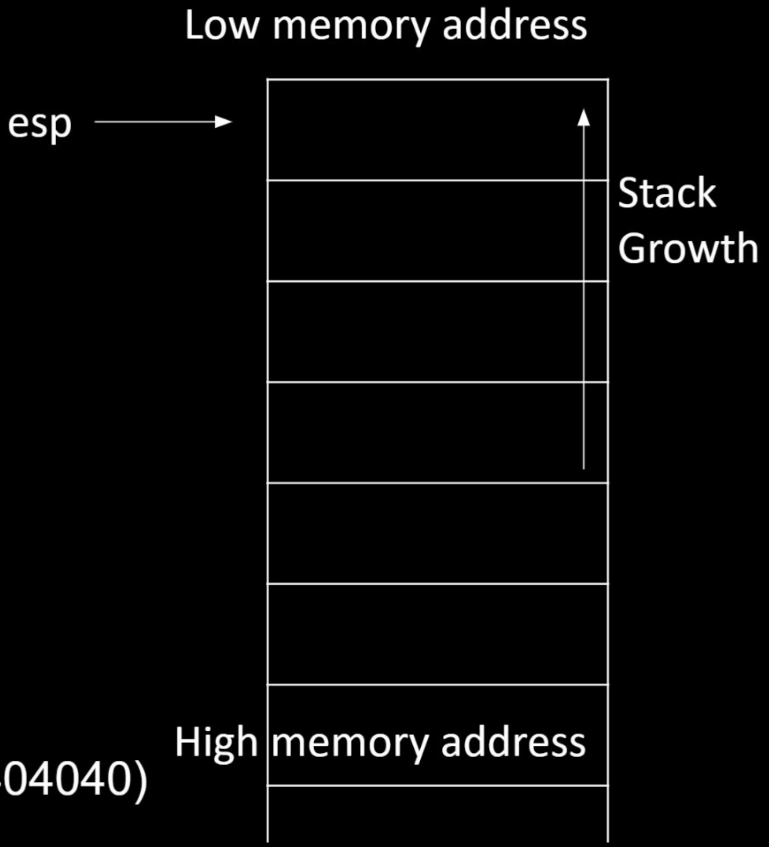
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`
- pop instruction reads the value on top of the stack and then adds to ESP
  - Example: `pop %eax`



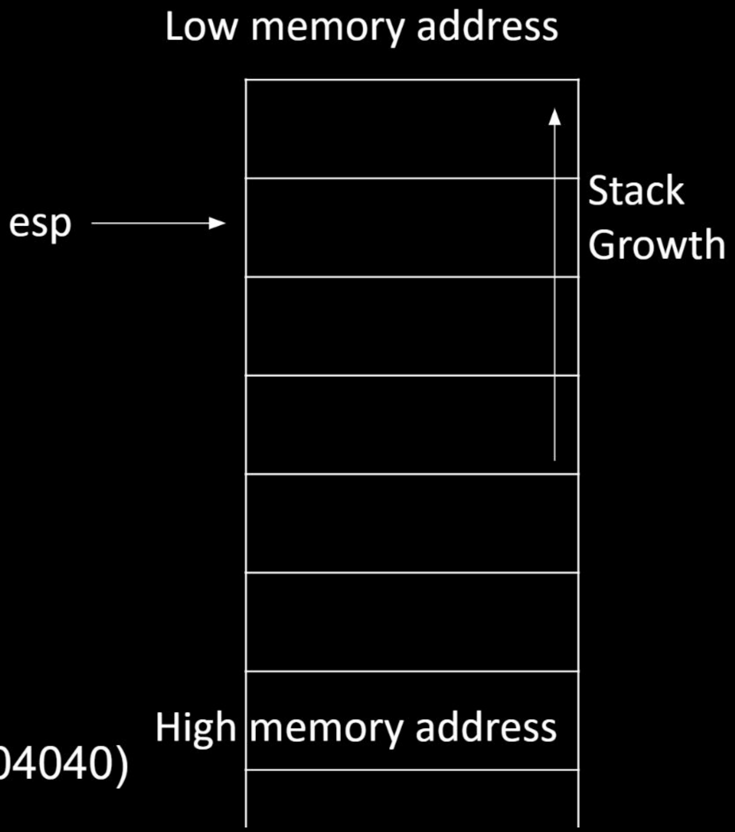
The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts from ESP and then writes to the top of the stack
    - Example: `push 0x40404040`
- pop instruction reads the value on top of the stack and then adds to ESP
    - Example: `pop %eax` (%eax ⇐ 0x40404040)



The Stack

- ESP (stack pointer) points to the top of the stack
- push instruction subtracts from ESP and then writes to the top of the stack
  - Example: `push 0x40404040`
- pop instruction reads the value on top of the stack and then adds to ESP
  - Example: `pop %eax` (%eax ⇐ 0x40404040)



Stack Frames

```
void bar {
   int a = 5;  // (push $5)
   int b = 10; //(push $10)
   foo(12,11);
}
```

Stack Frames

```
void bar {
   int a = 5;  // (push $5)
   int b = 10; //(push $10)
   foo(12,11);
}
```

```
bar:
   push %ebp
   mov  %esp, %ebp
   mov  $5,   %eax
   mov  $10,  %ebx
   push $11
   push $12
   call foo
   leave
   ret
```

Stack Frames

```
void bar {
   int a = 5;  // (push $5)
   int b = 10; //(push $10)
   foo(12,11);
}
```

```
bar:
   push %ebp
   mov  %esp, %ebp
   mov  $5,   %eax
   mov  $10,  %ebx
   push $11
   push $12
   call foo
   leave
   ret
```

Function Prologue
(Sets up stack frame)

Stack Frames

```
void bar {
   int a = 5;  // (push $5)
   int b = 10; //(push $10)
   foo(12,11);
}
```

```
bar:
   push %ebp
   mov  %esp, %ebp
   mov  $5,   %eax
   mov  $10,  %ebx
   push $11
   push $12
   call foo
   leave
   ret
```

Function Epilogue
(Tear down stack frame and
return us to calling function)

Stack Frames

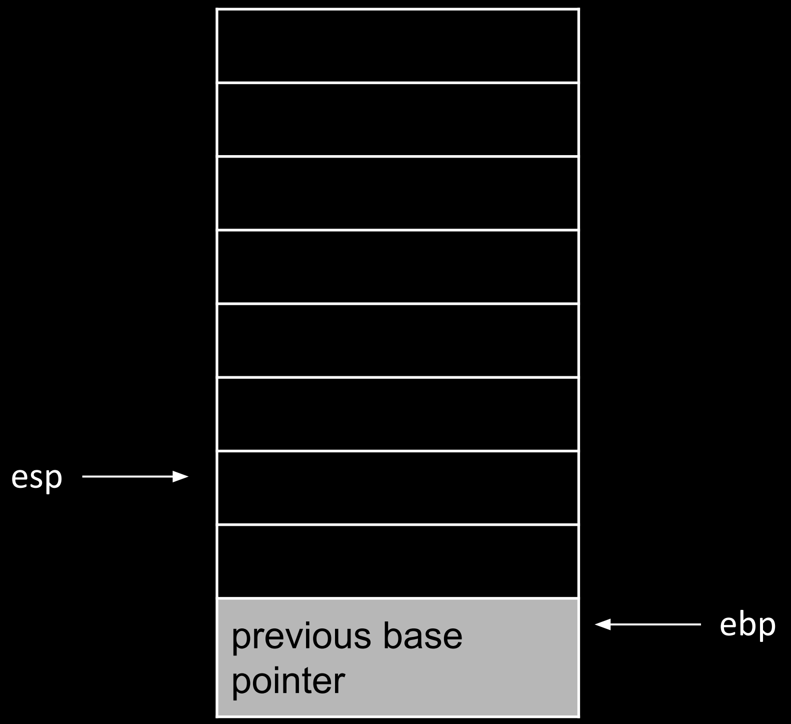
```
void bar {
   int a = 5;  // (push $5)
   int b = 10; //(push $10)
   foo(12,11);
}
```

```
bar:
   push %ebp
   mov  %esp, %ebp
   mov  $5,   %eax
   mov  $10,  %ebx
   push $11
   push $12
   call foo
   leave
   ret
```

Function Call
(Prepare arguments and
jump to another function)

Stack Frames

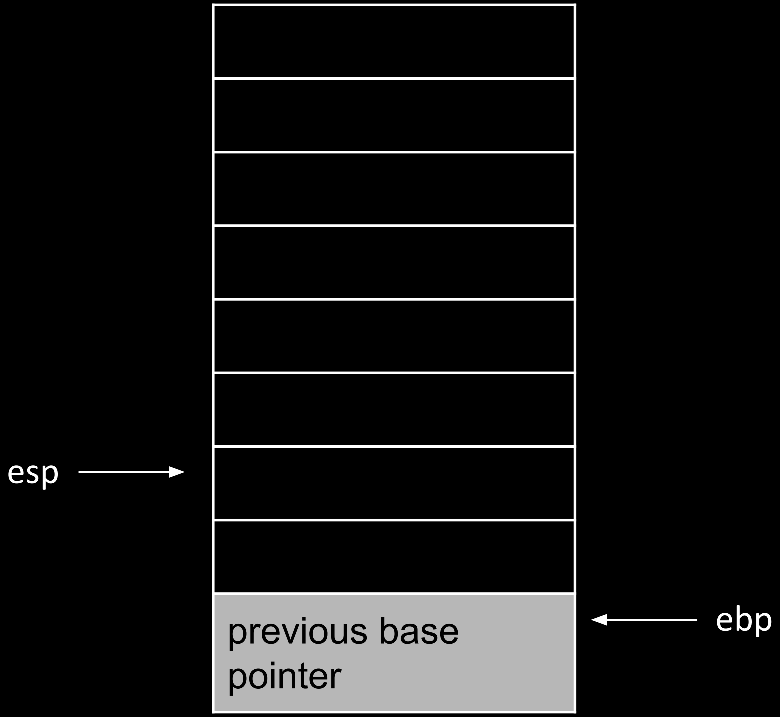
```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```



Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()

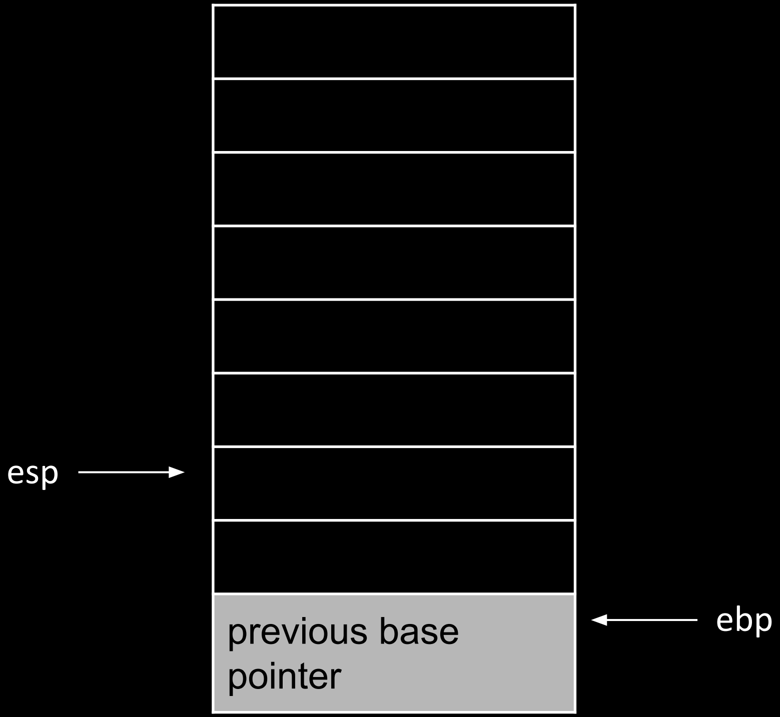


Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
   - Example: foo() takes 2 arguments, so we need to:

     `push $11, push $12`

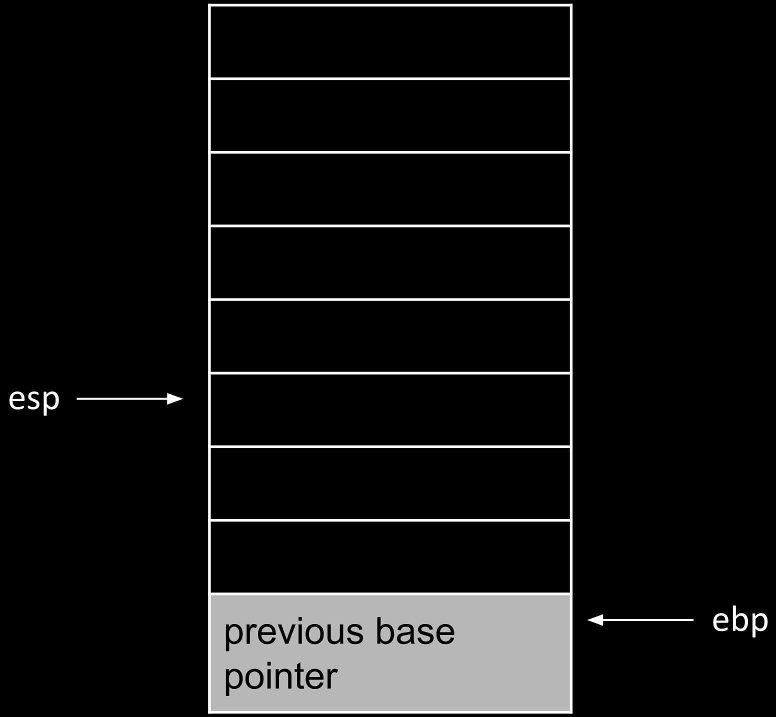


Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
    - ○ Example: foo() takes 2 arguments, so we need to:

```
push $11
```

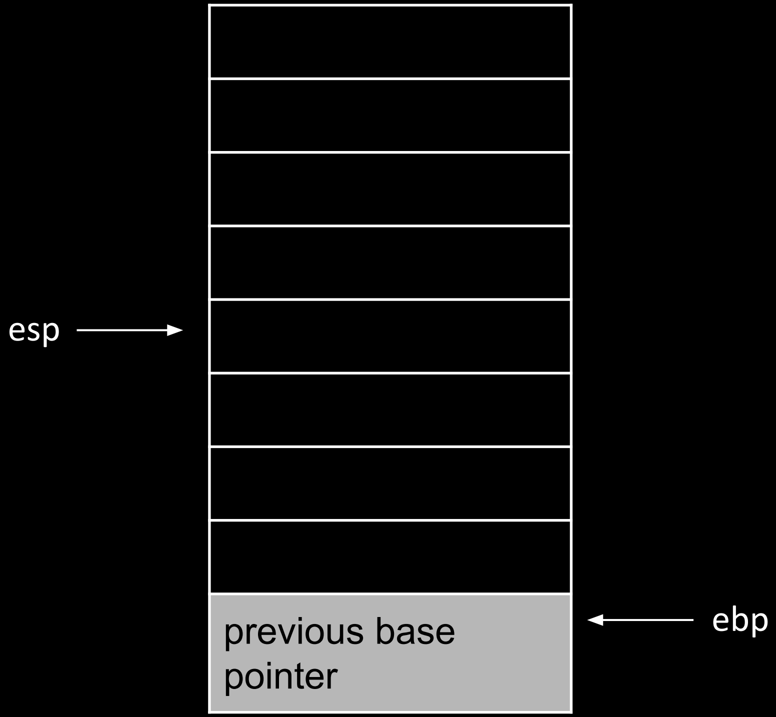


Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
    - ○ Example: foo() takes 2 arguments, so we need to:

      `push $11, push $12`



esp

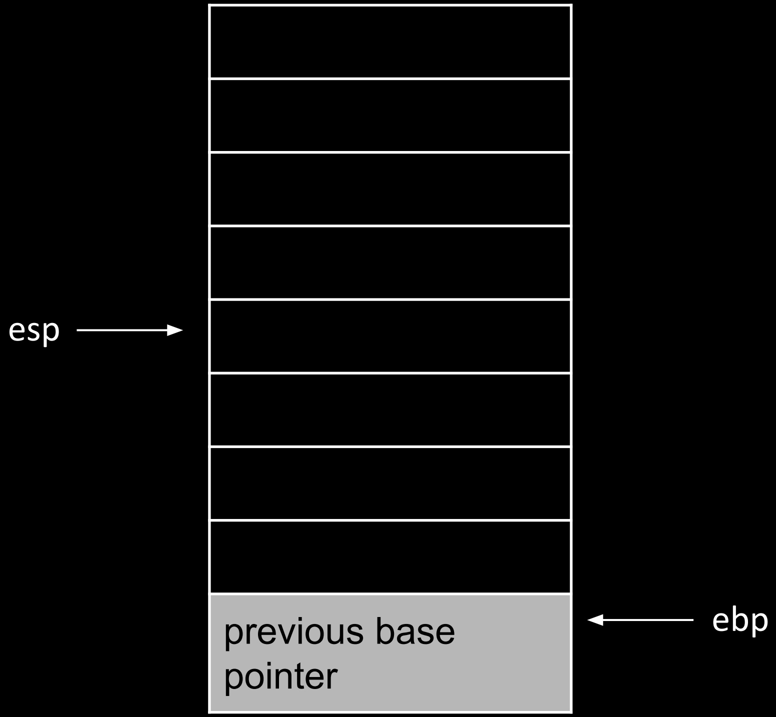
previous base pointer

ebp

Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

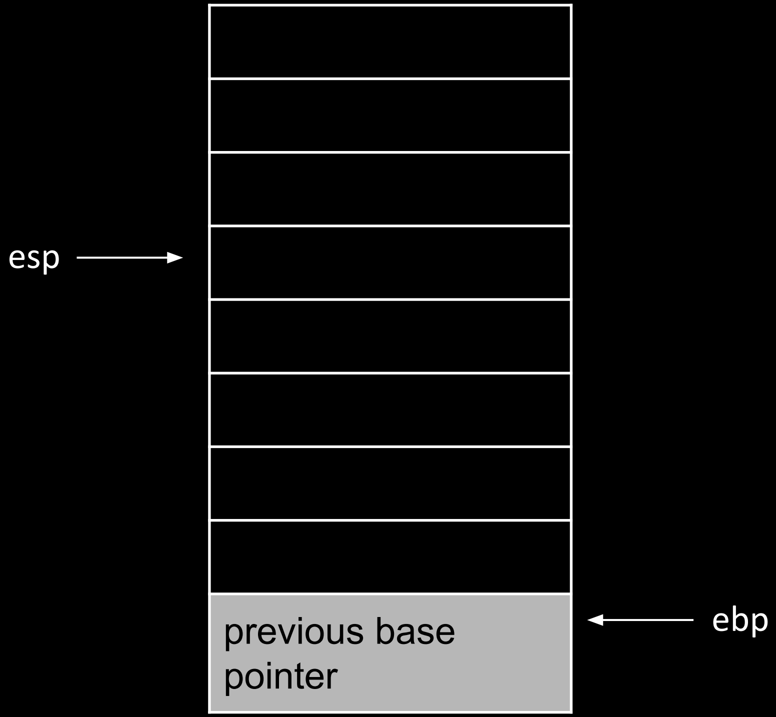
1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()



Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
    - call foo()
        - push EIP



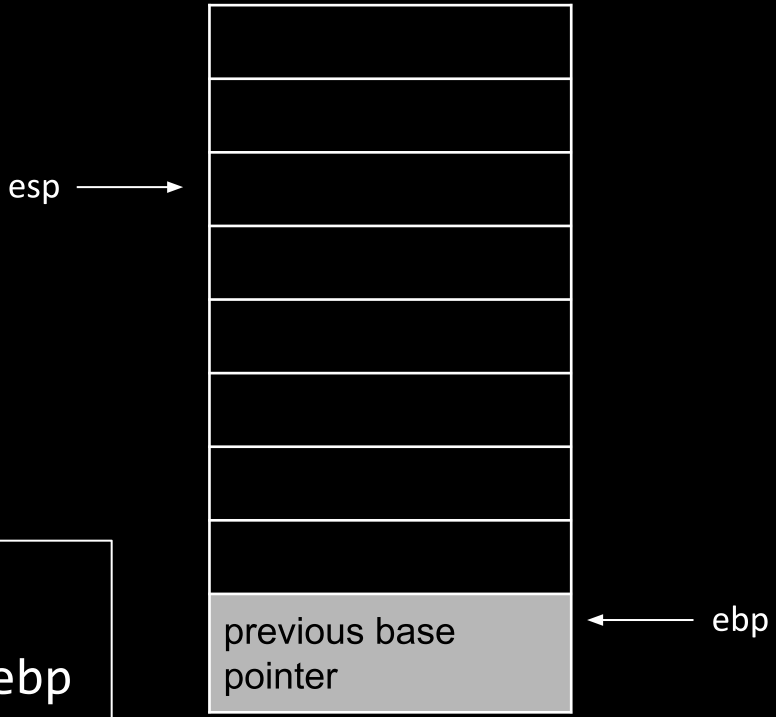
Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue

Function Prologue

```
foo:
 push %ebp
 mov %esp, %ebp
 ...
```



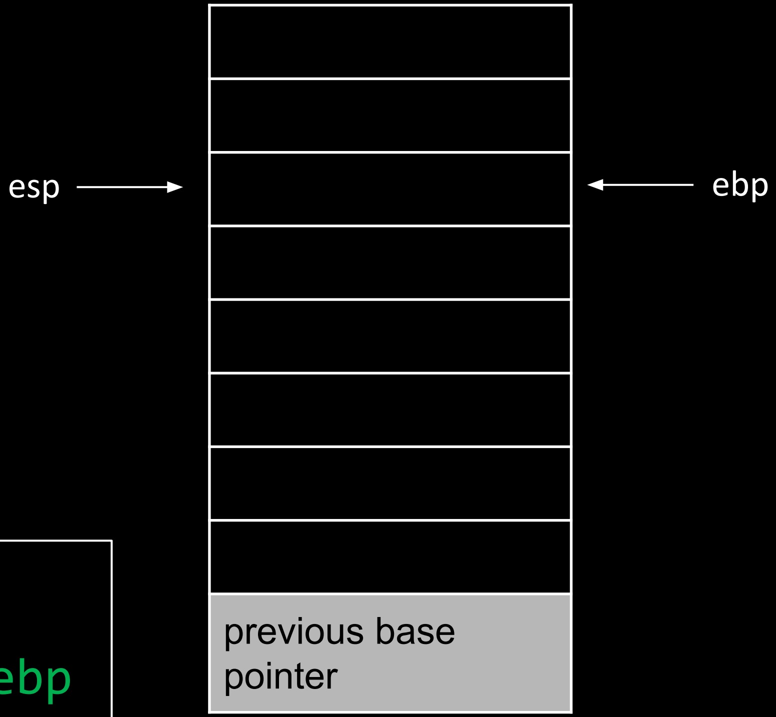
Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue

Function Prologue

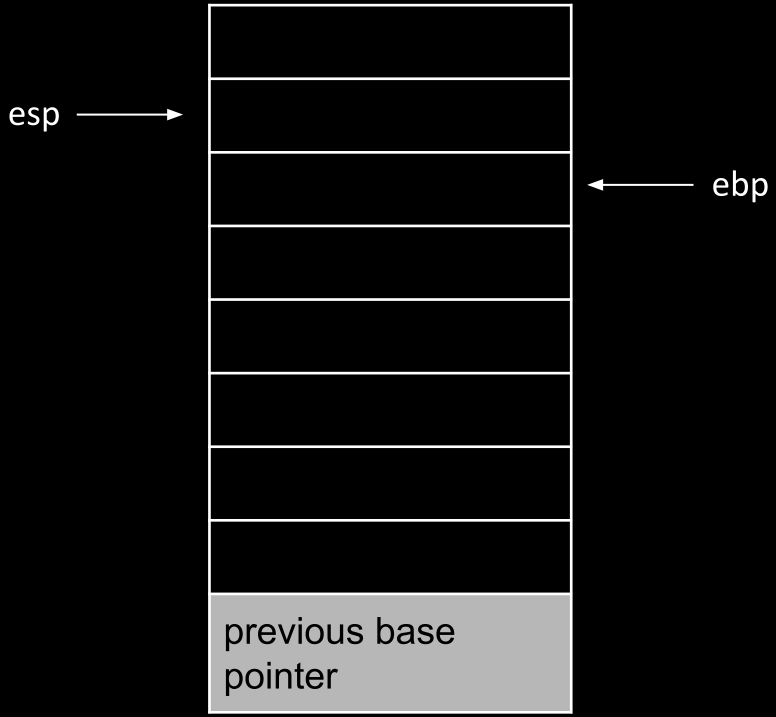
```
foo:
  push %ebp
  mov %esp, %ebp
  ...
```



Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

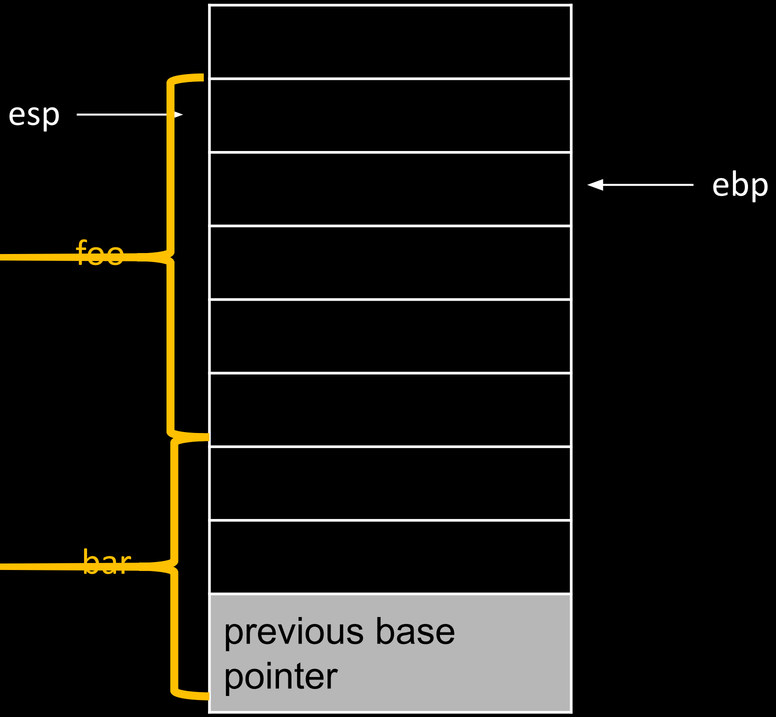
1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue
5. foo() local variables



Stack Frames

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue
5. foo() local variables



Stack Frames

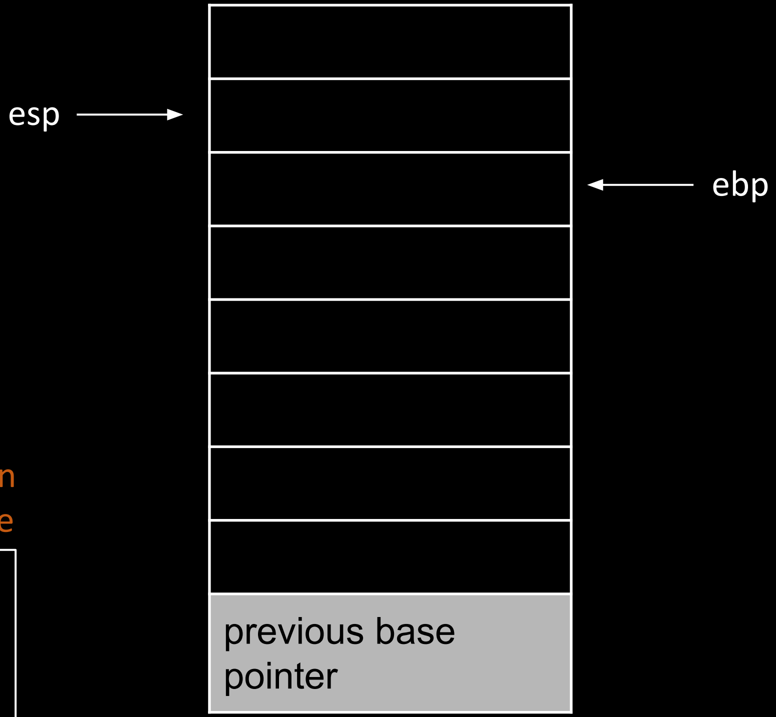
```
leave = mov %ebp, %esp
        pop %ebp
```

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue
5. foo() local variables
6. foo() epilogue

Function Epilogue

```
leave
ret
```



Stack Frames

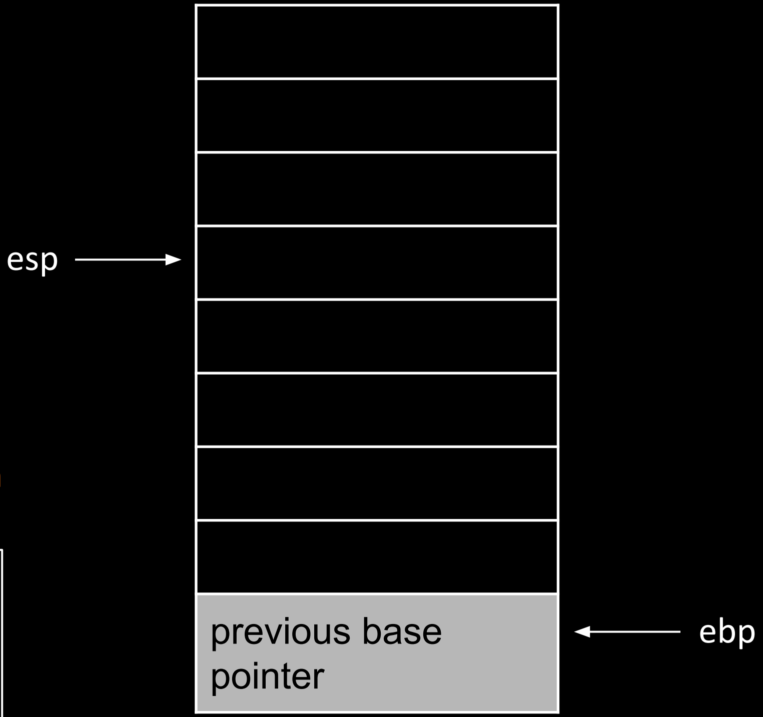
```
leave = mov %ebp, %esp
        pop %ebp
```

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue
5. foo() local variables
6. foo() epilogue

Function Epilogue

```
leave
ret
```



Stack Frames

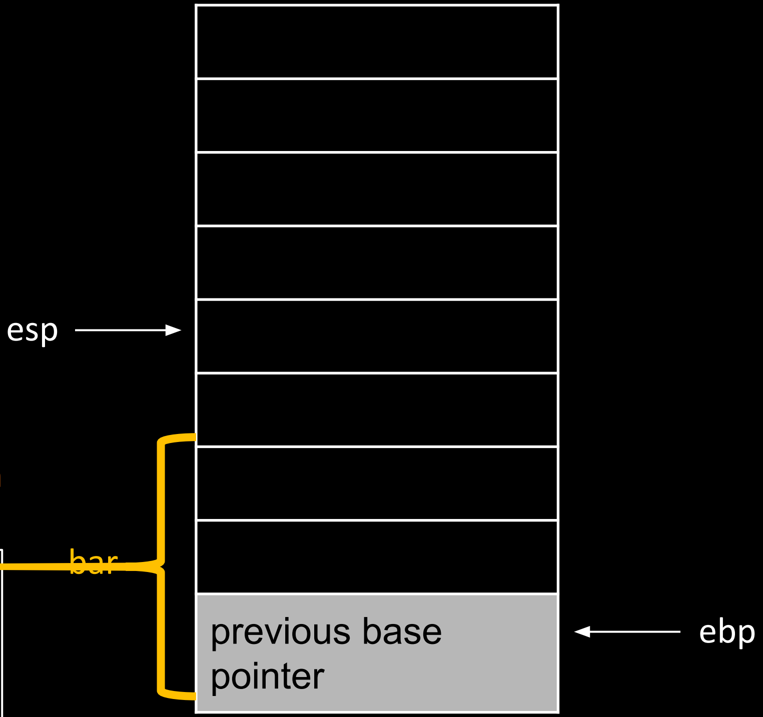
```
ret = pop %eip
```

```
void bar {
    int a = 5;  // (push $5)
    int b = 10; //(push $10)
    foo(12,11);
}
```

1. Do stuff in bar()
2. Set up arguments for foo()
3. Make a stack frame for foo()
4. foo() prologue
5. foo() local variables
6. foo() epilogue

Function Epilogue

```
leave
ret
```


esp
bar
previous base pointer
ebp