CSC2/452 Computer Organization
Functions

Sreepathi Pai

URCS

September 28, 2022

Outline

Outline

Assignments and Homeworks

- Assigment #2, Part I out today, due next Fri Oct 7.
  - Use the autograder to test your assignments
  - No penalties for incorrect submissions before deadline
  - Unlimited submissions before deadline
- Homework #4 out
  - Homework #4 due next MONDAY (Oct 3) in CLASS

Outline

Previously: Assembly Language

- Storage: Only memory or registers
  - Memory: heap or stack
- Registers: Limited storage on CPU
- Lack block structure
  - `goto`'s galore!
- Variables, Expressions, Conditionals, Loops

Outline

Basics of translating HLLs to Assembly (so far)

- Simplify expressions
- Find locations for variables
- Destructure loops
  - Use conditional and unconditional jumps
- Functions?

Handling Function Calls

- How to pass arguments to function?
- How to jump to a function?
- How to come back to just after call location?
  - How does `ret` know where to return to?
- How to receive the return value from a function?

How to pass arguments to functions

```
int sum3(int a, int b, int c) {
  int s = 0;

  s = a + b + c;

  return s;
}

int main(void) {
  int d = 3;

  sum3(1, 2, d);
}
```

- Parameters are variables
  - Where can we store variables?

Parameter Passing: Locations

- Memory
- Registers

Outline

Parameter Passing in Memory

```
int sum3(int a, int b, int c) {
  int s = 0;

  s = a + b + c;

  return s;
}

int main(void) {
  int d = 3;

  sum3(1, 2, d);
}
```

- Where shall we put the parameters in memory?
  - Both `main` and `sum` must agree on location!

Location, Location, Location

Which of the following is a good choice to store arguments to `sum`?

- A fixed, pre-determined, reserved location in memory
  - Say address `0x10203040`
- A fixed, pre-determined, reserved location in memory for each function
- A new location for each function call, created by `main`
  - How does `sum` know where this location is?
- A new location (#1) for each function call, created by `main`
  - *and* a fixed location (#2) to send the address of the location #1
  - Where should location #2 be?

Design constraints for storing in memory

- Both caller and callee must agree on location
  - caller (e.g., `main`) to store arguments
  - callee (e.g., `sum3`) to access them
- When storing arguments in memory, other things caller and callee must also agree on:
  - Order of arguments in memory: left-to-right, or right-to-left?
  - I.e., `a`, `b`, `c` or `c`, `b`, `a`?
- How to pass *number* of arguments?
  - e.g., `printf("Today is %d %d %d", day, month, year);`
- Do we need to pass sizes of each argument?
- Who frees allocated region?
  - Caller? Callee?

Performance

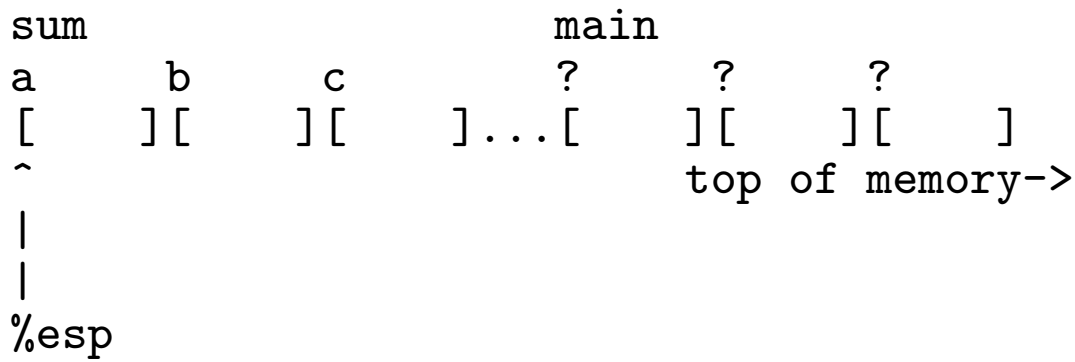
- Must be fast
  - New space must be allocated for each function call
  - This space must be released at the end of each function call
- Can't go looking around in memory for free space at every function call!

The Stack Frame

```
partial stack frame for main(void)

?    ?
[   ][   ]
partial stack frame for sum3

a     b     c     ?
[    ][    ][    ][    ]
```

- A region of memory created for each function call
- Created on function call
  - All arguments are placed into the frame by caller
  - Address of the frame passed to callee in a well-known pre-determined register
- Destroyed on function exit
- Also called the “call frame”

The Stack Frame on x86



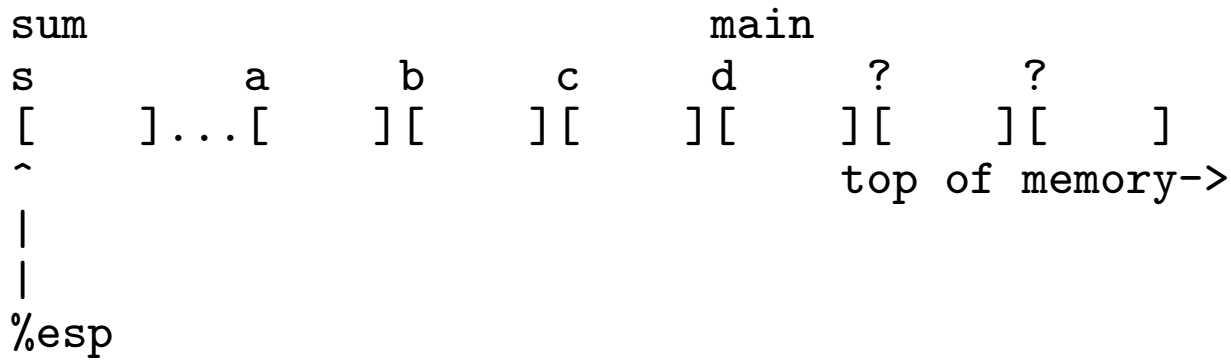
- Note: Each [ ] represents contiguous memory locations containing one 32-bit value
- Created in a region of memory called the stack
- On x86, %esp points to top of stack
  - %rsp in 64-bit mode
  - Where you can push new data
  - I.e., where you can allocate space!
- Note: stack grows downward on x86
- All call frames are next to each other in memory

Recursion

```
int sum_upto(int n) {
    int s = 0;

    if(n == 0) return 0;

    s = n + sum_upto(n - 1);

    return s;
}
```

- We know each call of `sum_n` gets its own copy of `n`
  - Because `n` is a parameter and is part of the stack frame
- What about local variable `s`?
  - Is it local to `sum_upto` or to each call of `sum_upto`?

Another use for the stack frame: Storing locals

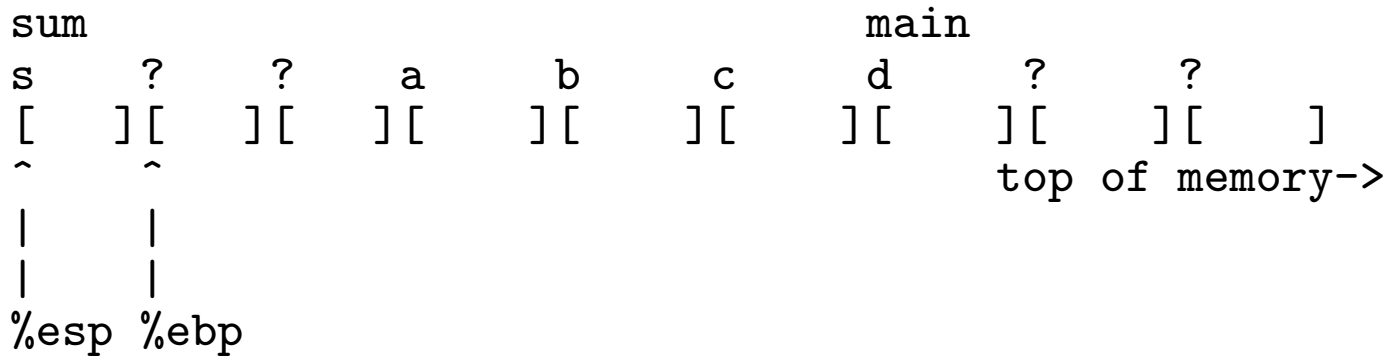


- Since locals are local to a function *call*
  - each function call gets its own copy
- They are also stored on the stack frame for a function call
- How to address them?
  - %esp only points to top of stack
  - And can change as we push/pop

The base pointer

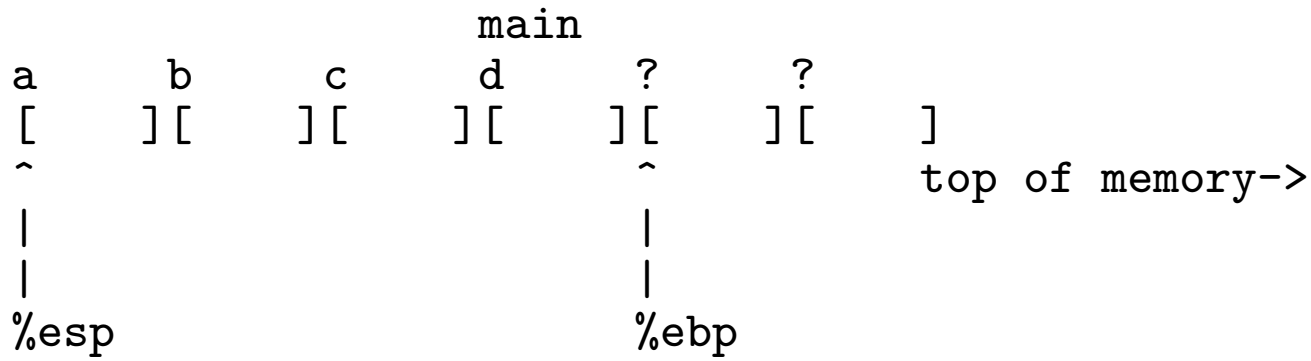
- The base pointer contains the address of the *base* of the stack frame
  - The address *after* caller has pushed arguments
  - But before *callee* has pushed locals
- On x86-64, the base pointer is `%ebp`
  - The callee copies `%esp` (caller's top-of-stack) into `%ebp` on entry
  - i.e., `mov %esp, %ebp`
- It then subtracts `%esp` to make space for locals
  - `subl 0x4, %esp`

Addressing via the base pointer

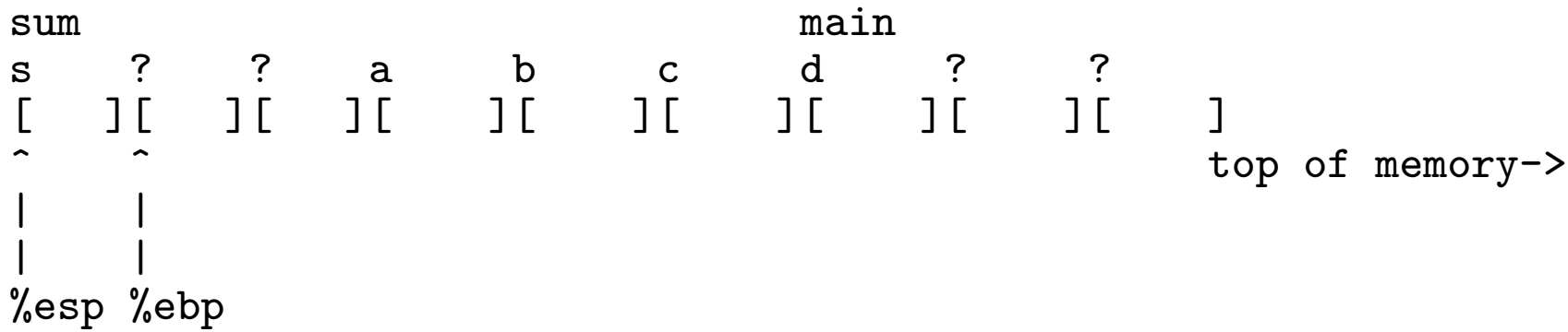


- Arguments:
  - a: `0x8(%ebp)`
  - b: `0xc(%ebp)`
  - c: `0x10(%ebp)`
- Locals
  - s: `-0x4(%ebp)`

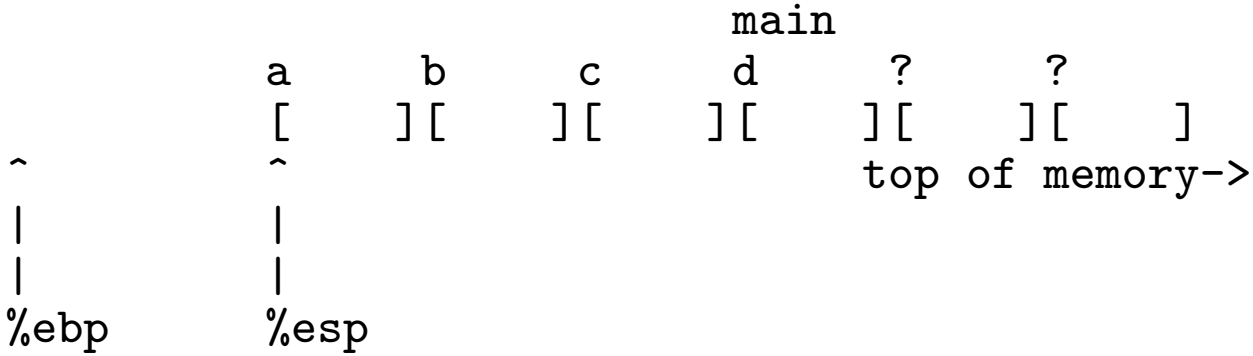
Stack frame on x86: before calling `sum`



Stack frame on x86: after calling sum



What happens when `sum` returns?

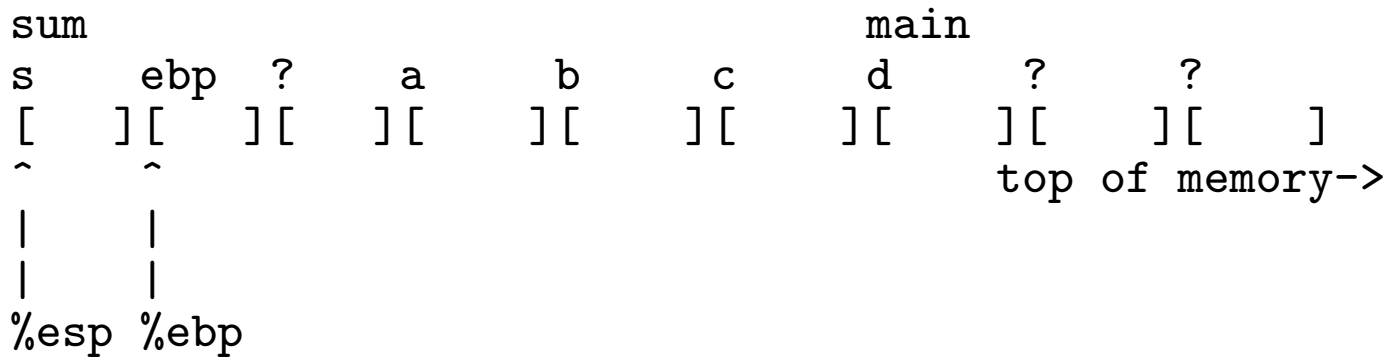


- If we just pop off locals, and mystery objects, %esp is restored
- But what about %ebp?
  - How do we restore the value of %ebp for `main` after `sum` returns?
  - `main` won't be able to access its locals or arguments!

Prologue and Epilogue on x86

- Prologue:
  - callee saves caller's base pointer on stack: `push %ebp`
  - callee copies caller's top-of-stack `%esp` into `%ebp` to establish its base pointer: `mov %esp, %ebp`
- Epilogue
  - callee restores caller's top-of-stack: `mov %ebp, %esp`
  - callee restores caller's base pointer from stack: `pop %ebp`
  - on x86, a single instruction does both: `leave`

What happens when `sum` returns?

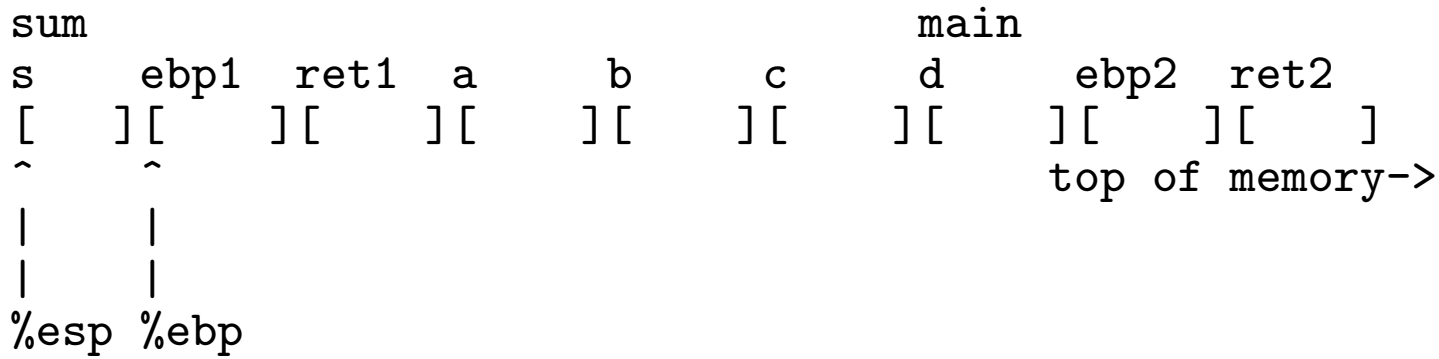


- The `ebp` labeled location contains the value of `main`'s %ebp
- Save there by `sum` executing `push %ebp` when it begins

How does `ret` know where to return to?

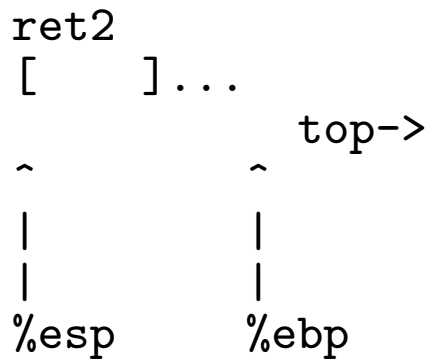
- The return address is also specific to each function call
- Where should `call` store the return address?

Complete stack frame



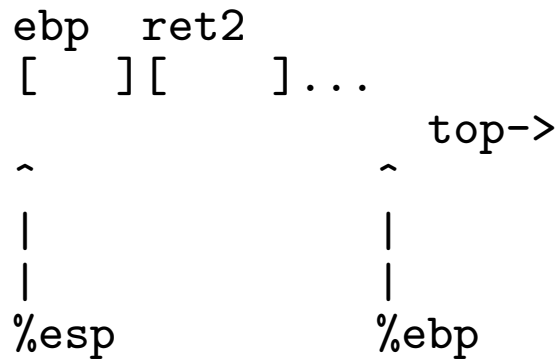
- `retX` contains the return address specific to each function call
- Pushed to the stack by `call`
- Value is the address of the instruction *after* call

The whole story: Calling sum



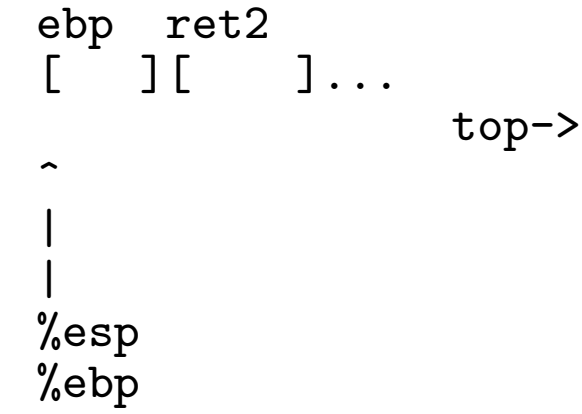

```
0000050f <main>:
*50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
```

The whole story: Calling sum



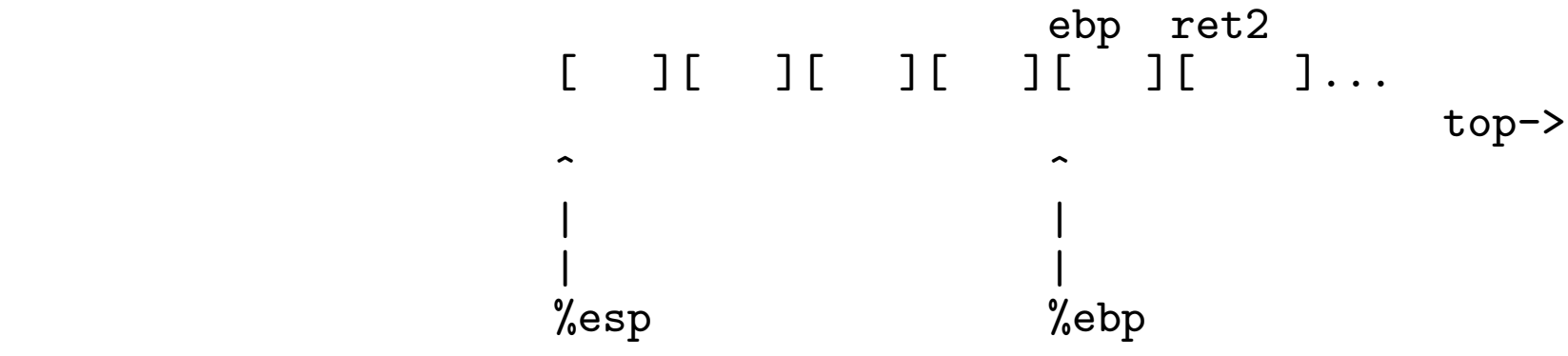

```
0000050f <main>:
 50f:   55                      push   %ebp
*510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
```

The whole story: Calling sum



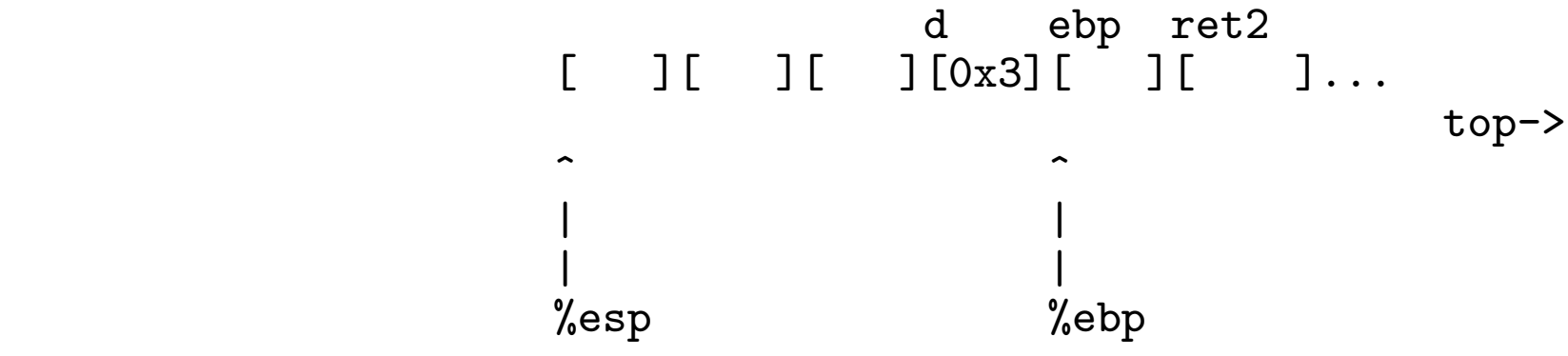

```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
*512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
```

The whole story: Calling sum



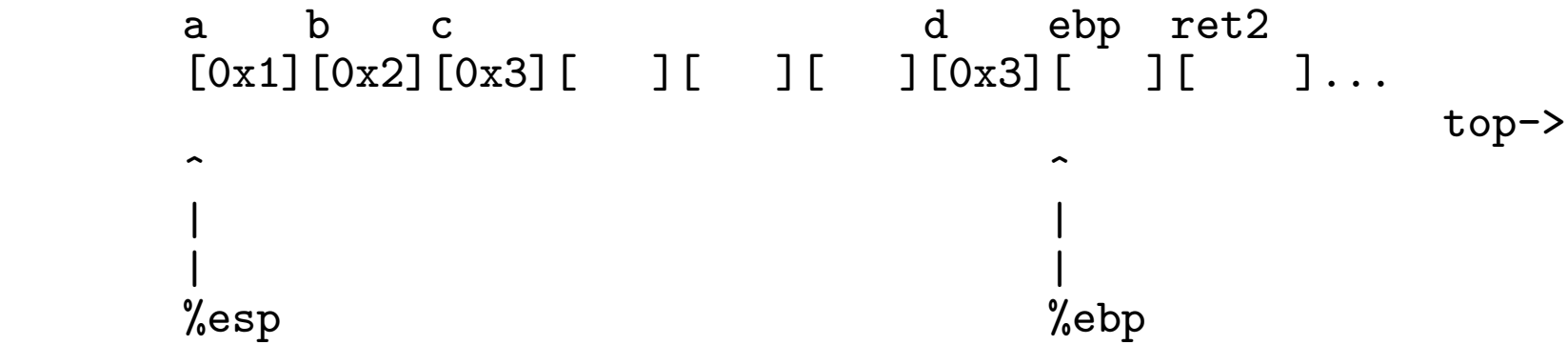

```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
*515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
```

The whole story: Calling sum




```
0000050f <main>:
 50f:   55                          push   %ebp
 510:   89 e5                       mov    %esp,%ebp
 512:   83 ec 10                    sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00        movl   $0x3,-0x4(%ebp)
*51c:   ff 75 fc                    pushl  -0x4(%ebp)
 51f:   6a 02                       push   $0x2
 521:   6a 01                       push   $0x1
 523:   e8 c5 ff ff ff              call   4ed <sum3>
 528:   83 c4 0c                    add    $0xc,%esp
```

The whole story: Calling sum




```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
*523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
```

The whole story: Calling sum

```
  ret1 a    b    c                    d    ebp  ret2
  [528][0x1][0x2][0x3][   ][   ][   ][0x3][   ][    ]...
                                                  top->
  ^                                    ^
  |                                    |
  |                                    |
  %esp                                 %ebp
```

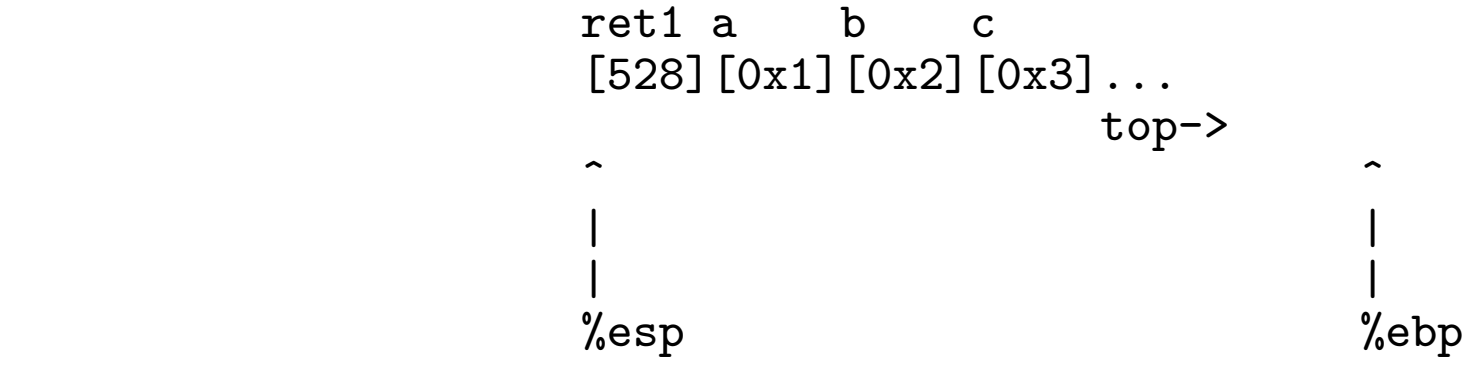
```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
```

Inside sum: Just before leave

```
              s     ebp  ret1 a    b    c
[   ][   ]..[   ][   ][528][0x1][0x2][0x3]
                                        top->
^             ^     ^          ^    ^    ^
|             |     |          |    |    |
|             |     |          |    |    |
%esp          -0x4 %ebp       +0x8 +0xc +0x10
```

```
000004ed <sum3>:
 4ed:   55                      push   %ebp
 4ee:   89 e5                   mov    %esp,%ebp
 4f0:   83 ec 10                sub    $0x10,%esp
 4f3:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 4fa:   8b 55 08                mov    0x8(%ebp),%edx
 4fd:   8b 45 0c                mov    0xc(%ebp),%eax
 500:   01 c2                   add    %eax,%edx
 502:   8b 45 10                mov    0x10(%ebp),%eax
 505:   01 d0                   add    %edx,%eax
 507:   89 45 fc                mov    %eax,-0x4(%ebp)
 50a:   8b 45 fc                mov    -0x4(%ebp),%eax
*50d:   c9                      leave
 50e:   c3                      ret
```

Inside sum: Just after leave



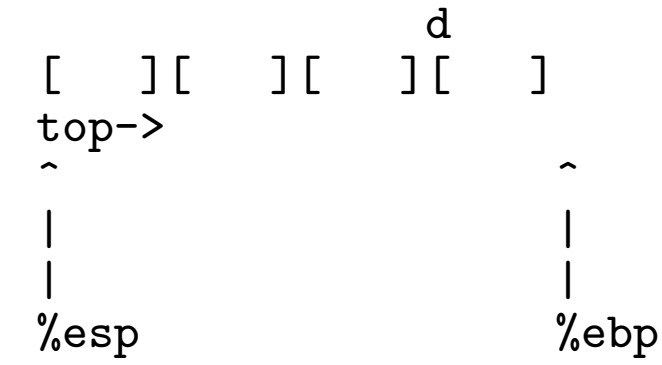

```
000004ed <sum3>:
 4ed:   55                      push   %ebp
 4ee:   89 e5                   mov    %esp,%ebp
 4f0:   83 ec 10                sub    $0x10,%esp
 4f3:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 4fa:   8b 55 08                mov    0x8(%ebp),%edx
 4fd:   8b 45 0c                mov    0xc(%ebp),%eax
 500:   01 c2                   add    %eax,%edx
 502:   8b 45 10                mov    0x10(%ebp),%eax
 505:   01 d0                   add    %edx,%eax
 507:   89 45 fc                mov    %eax,-0x4(%ebp)
 50a:   8b 45 fc                mov    -0x4(%ebp),%eax
 50d:   c9                      leave
*50e:   c3                      ret
```

Back in main

```
                                a    b    c
                                [0x1][0x2][0x3]...
                                                top->
                                ^                    ^
                                |                    |
                                |                    |
                                %esp                 %ebp
```

```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
*528:   83 c4 0c                add    $0xc,%esp
```

Back in main




```
0000050f <main>:
 50f:   55                      push   %ebp
 510:   89 e5                   mov    %esp,%ebp
 512:   83 ec 10                sub    $0x10,%esp
 515:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%ebp)
 51c:   ff 75 fc                pushl  -0x4(%ebp)
 51f:   6a 02                   push   $0x2
 521:   6a 01                   push   $0x1
 523:   e8 c5 ff ff ff          call   4ed <sum3>
 528:   83 c4 0c                add    $0xc,%esp
* ...
```

`leave` and `ret`

- Recall `leave` is:
  - `mov %ebp, %esp`
  - `pop %ebp`
- Then, `ret` just pops off the return address into `%eip`

Summary: Calling convention

- Function calls setup stack frames
  - On x86, delimited by addresses `%esp` (top) and `%ebp` (base)
- Arguments are passed on the stack in region shared by caller and callee
- Return address is stored on the stack as well
- Callee setups up stack frame for local variables by initializing base pointer to current top of stack
- Callee restores caller's stack frame on return
  - I.e., values of caller's `%ebp` and `%esp`
- Then continues execution at address stored on stack

C Calling Convention (`cdecl`)

- Arguments pushed on to stack right to left
- Allows *varargs* functions like `printf` to work
  - `printf` does not know how many arguments have been pushed
  - It reads the format string to figure this out
  - Therefore format string must be `+8(%ebp)` (i.e. pushed last)
- Callee does not know how many arguments have been pushed
- Caller frees stack space for arguments
  - Just an addition instruction

Pascal Calling Convention

- Arguments pushed on to stack left to right
- Does not support varargs
- Callee frees stack space for arguments
- Variant: `stdcall`
  - Windows API, pushes right to left like `cdecl`
  - But callee cleans up stack space

Why all %e registers?

- On x86, passing parameters on the stack is used largely by 16-bit and 32-bit code
- Therefore, the 32-bit registers %e*

Outline

Passing parameters on the stack can be slow

- Need to access memory
- Need to use indirect addresses to read/write
- Need to manipulate `%ebp` and `%esp`
- Why not use registers?
  - x86 originally only had 8 general-purpose registers
  - but x64 has 16!

The Application Binary Interface

- The ABI is a set of "rules" or conventions on a number of things
- One of them happens to be parameter passing
  - The System V x86-64 ABI describes how to pass registers on the x86-64
  - Different for each processor
- However, return address continues to be on the stack
- Locals may also be on the stack
- A stack frame is still setup for each function
- Reading for this week

Ramifications

What happens if you overwrite the return address on the stack?

- It is in memory, and can be written to directly
  - `mov some-value, 4(%ebp)` (assuming 32-bit system)
- Can be done using buffer overflows
  - Was used by the Morris Worm in 1988

Functions and Function Instances

- A function is piece of code
- A running function is an function instance
  - Function + Stack Frame = Function Instance

References

- Chapter 3 of the textbook
- After Fall Break:
  - C Pointers
  - Structures, Unions, etc.