Announcements

assign4 due Tuesday

Midterm this Friday

- See website for location and practice exam

Assembly Roadmap

Last Week

- Registers, addressing modes, mov
- Arithmetic and logical operations
- Control structures (if/else, loops)

Today

- Using the stack
- Function call and return

Goals for Today

Using the stack for local variables

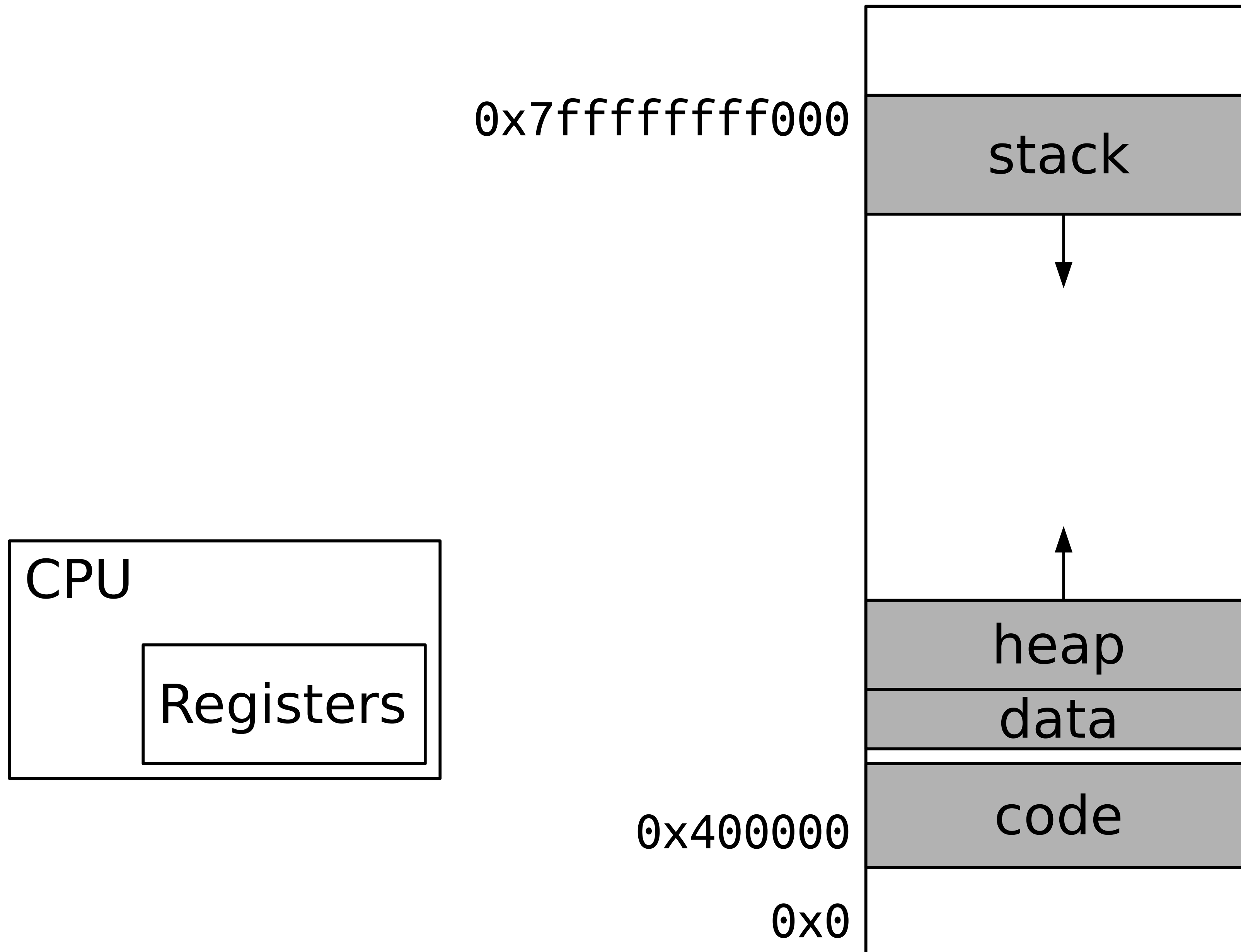
- How the stack works, how to manage it

Function call and return in assembly

- Instructions, calling convention

Using the stack to save registers

CPU and Memory



Uses for the Stack

Local variables

- More variables than fit in registers
- Arrays, large structs
- Variables which need an address

Temporary space for state

- Save info across function calls

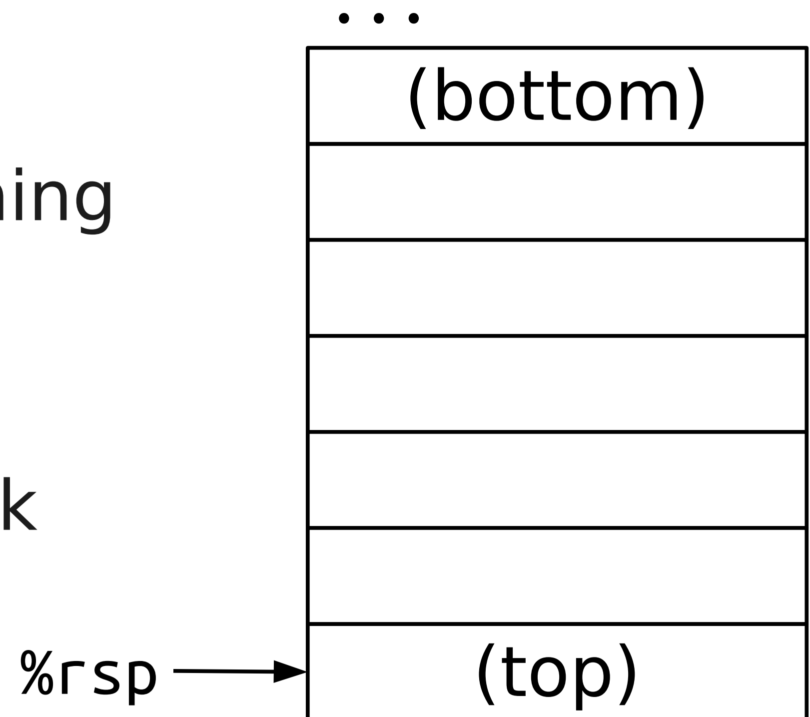
Stack Essentials

Grows downward (toward lower addresses)

- Top of the stack (most recent thing added) at lowest address

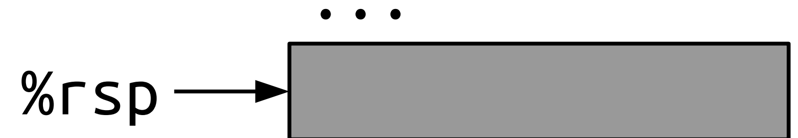
`%rsp`: stack pointer

- Reserved to point to top of stack
- Stores an address
- Can use normal arithmetic ops



Stack Array

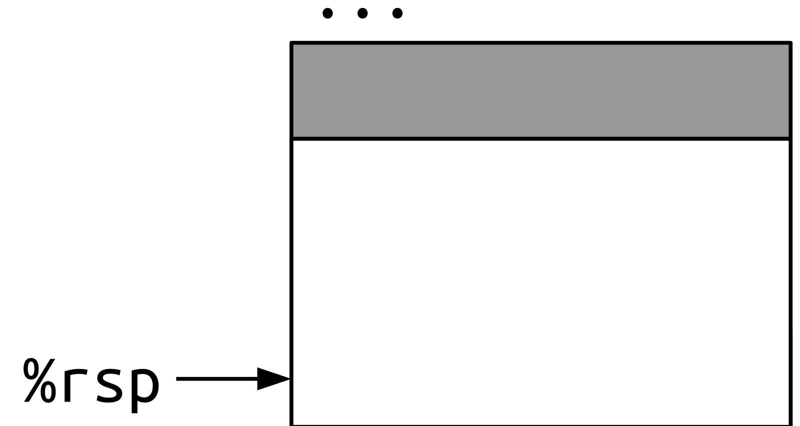
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
         stack_arr:
0x400594   sub    $0x18,%rsp
0x400598   movl   $0x0,(%rsp)
0x40059f   movl   $0x1e,0xc(%rsp)
           ...
0x4005b4   add    $0x18,%rsp
0x4005b8   retq
```

Stack Array

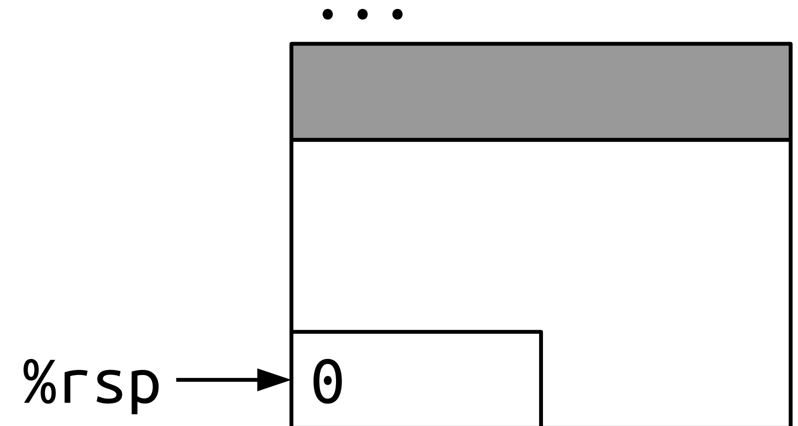
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
          stack_arr:
0x400594    sub     $0x18,%rsp
0x400598    movl    $0x0,(%rsp)
0x40059f    movl    $0x1e,0xc(%rsp)
            ...
0x4005b4    add     $0x18,%rsp
0x4005b8    retq
```

Stack Array

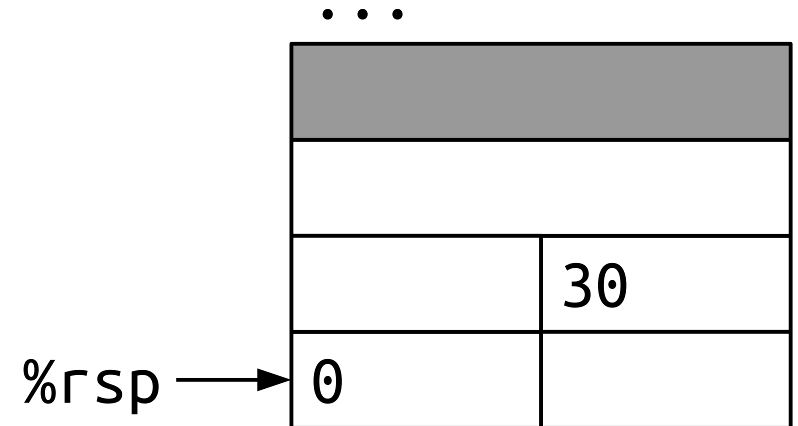
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
         stack_arr:
0x400594   sub     $0x18,%rsp
0x400598   movl    $0x0,(%rsp)
0x40059f   movl    $0x1e,0xc(%rsp)
           ...
0x4005b4   add     $0x18,%rsp
0x4005b8   retq
```

Stack Array

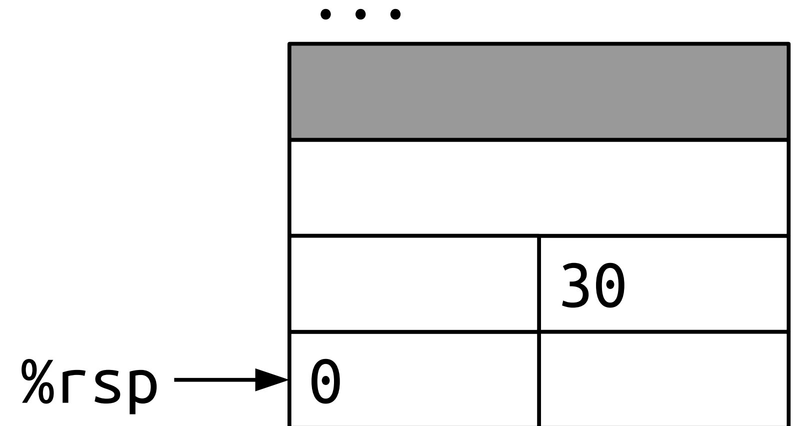
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
          stack_arr:
0x400594    sub     $0x18,%rsp
0x400598    movl    $0x0,(%rsp)
0x40059f    movl    $0x1e,0xc(%rsp)
            ...
0x4005b4    add     $0x18,%rsp
0x4005b8    retq
```

Stack Array

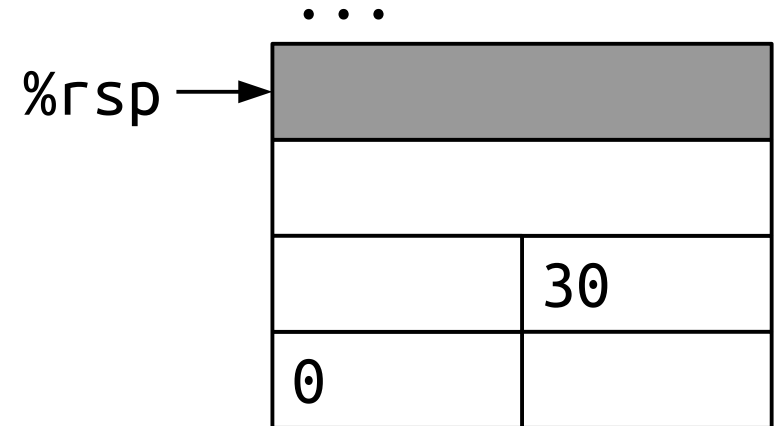
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
          stack_arr:
0x400594    sub     $0x18,%rsp
0x400598    movl    $0x0,(%rsp)
0x40059f    movl    $0x1e,0xc(%rsp)
            ...
0x4005b4    add     $0x18,%rsp
0x4005b8    retq
```

Stack Array

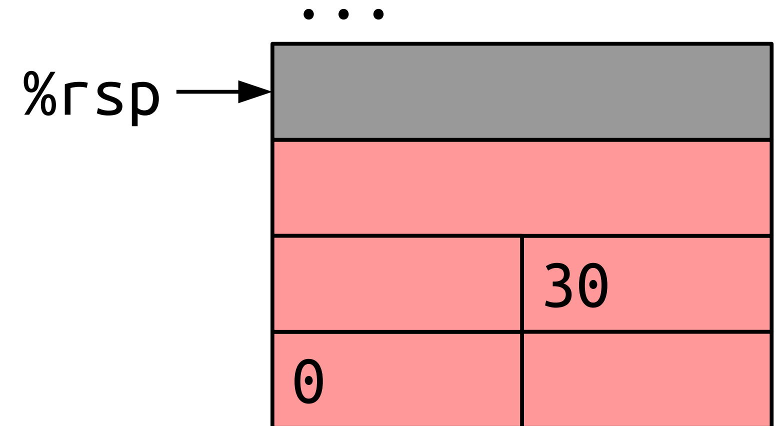
```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
         stack_arr:
0x400594   sub    $0x18,%rsp
0x400598   movl   $0x0,(%rsp)
0x40059f   movl   $0x1e,0xc(%rsp)
           ...
0x4005b4   add    $0x18,%rsp
0x4005b8   retq
```

Stack Array

```
void stack_arr() {
  int arr[4];
  arr[0] = 0;
  arr[3] = 30;
  ...
}
```




```
        stack_arr:
0x400594   sub     $0x18,%rsp
0x400598   movl    $0x0,(%rsp)
0x40059f   movl    $0x1e,0xc(%rsp)
           ...
0x4005b4   add     $0x18,%rsp
0x4005b8   retq
```

Observations

- **Allocate space by subtracting from `%rsp`**
  - "Cheap": just an arithmetic operation
  - OK to overallocate a little
- **Must deallocate space by adding to `%rsp`**
  - Leave stack pointer where it was at function entry
- **Memory not cleared after function return**

Function Calls

Caller

- Put arguments into registers (`%rdi`, `%rsi`, ...)
- Take note of where to return to
- Transfer control to callee

Callee

- (Optional) Allocate space on stack
- Do something (using parameters in registers)
- Restore stack to original state (if needed)
- Return control to caller

Caller

- Read return value from `%rax`

Function Calls

Caller

- Put arguments into registers (`%rdi`, `%rsi`, ...)
- Take note of where to return to
- Transfer control to callee

Callee

- (Optional) Allocate space on stack
- Do something (using parameters in registers)
- Restore stack to original state (if needed)
- Return control to caller

Caller

- Read return value from `%rax`

Instructions for Function Calls

`call [addr]`: Jump to the start of a function

`ret`: Return from a function

(q suffix optional)

```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
           id:
0x4005b9     mov     %edi,%eax
0x4005bb     retq
           fn_call:
0x4005bc     mov     $0x6b,%edi
0x4005c1     callq   0x4005b9 <id>
0x4005c6     add     $0x2a,%eax
0x4005c9     retq
```

Instructions for Function Calls

`call [addr]`: Jump to the start of a function

`ret`: Return from a function

- (q suffix optional)

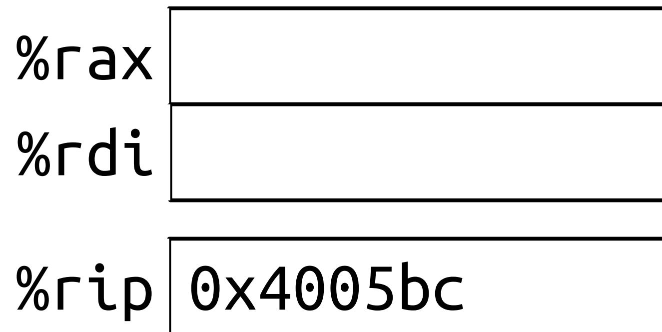
Difference between `call` and `jmp`

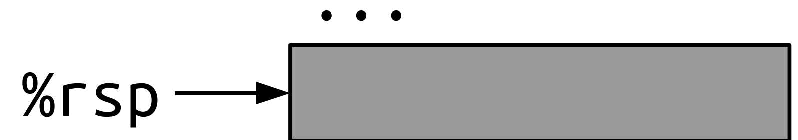
- `jmp` doesn't remember where we came from
- `call` stores the location to return to on the stack

The instruction pointer `%rip`

- Points to next instruction to execute
- Can't be manipulated like other registers

Function Call Example

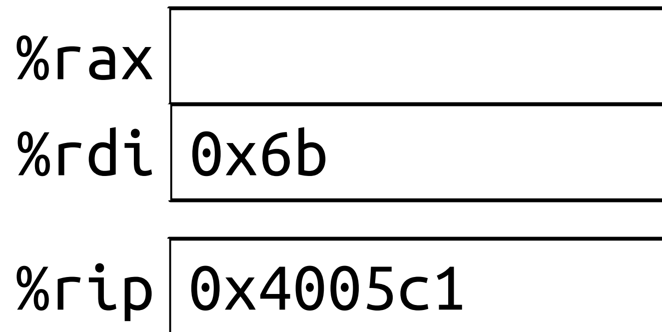


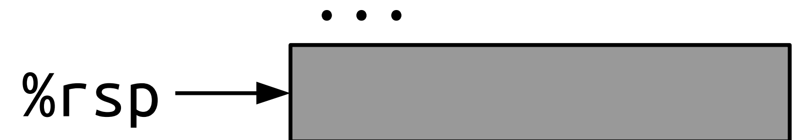



```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
          id:
0x4005b9    mov     %edi,%eax
0x4005bb    retq
          fn_call:
0x4005bc    mov     $0x6b,%edi
0x4005c1    callq   0x4005b9 <id>
0x4005c6    add     $0x2a,%eax
0x4005c9    retq
```

Function Call Example

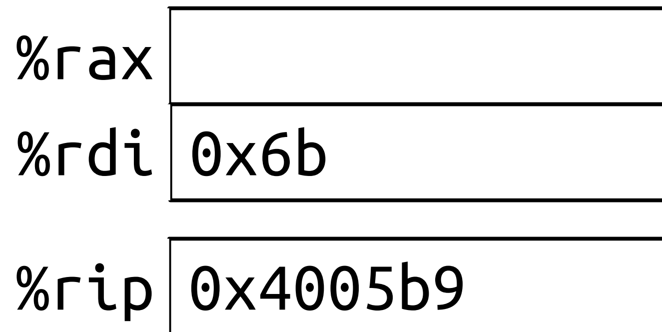


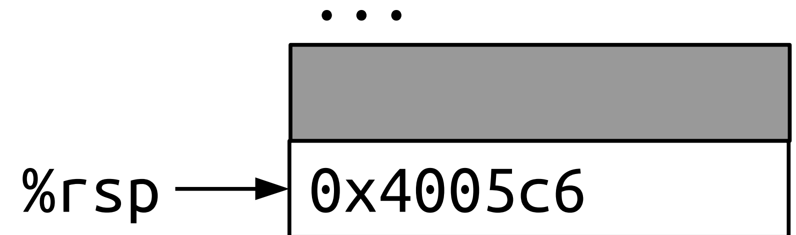



```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
          id:
0x4005b9    mov     %edi,%eax
0x4005bb    retq
          fn_call:
0x4005bc    mov     $0x6b,%edi
0x4005c1    callq   0x4005b9 <id>
0x4005c6    add     $0x2a,%eax
0x4005c9    retq
```

Function Call Example



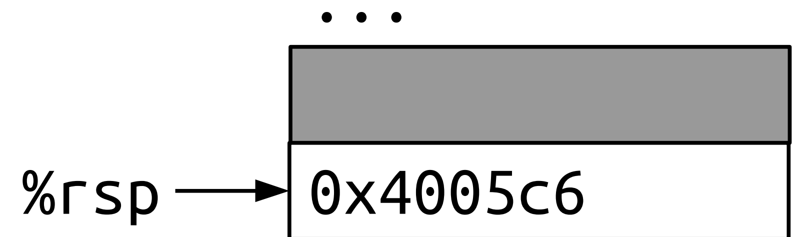



```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
          id:
0x4005b9    mov     %edi,%eax
0x4005bb    retq
          fn_call:
0x4005bc    mov     $0x6b,%edi
0x4005c1    callq   0x4005b9 <id>
0x4005c6    add     $0x2a,%eax
0x4005c9    retq
```

Function Call Example

%rax	0x6b
%rdi	0x6b
%rip	0x4005bb

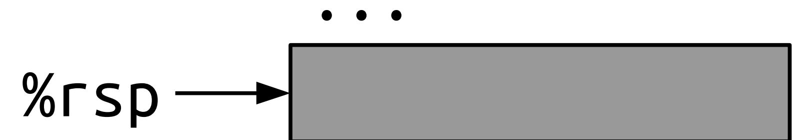



```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
         id:
0x4005b9   mov     %edi,%eax
0x4005bb   retq
         fn_call:
0x4005bc   mov     $0x6b,%edi
0x4005c1   callq   0x4005b9 <id>
0x4005c6   add     $0x2a,%eax
0x4005c9   retq
```

Function Call Example

%rax	0x6b
%rdi	0x6b
%rip	0x4005c6

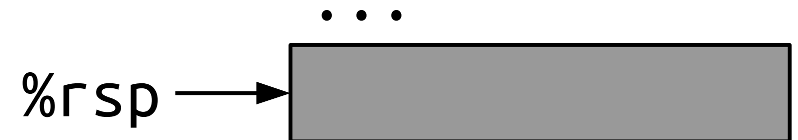



```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
          id:
0x4005b9    mov     %edi,%eax
0x4005bb    retq
          fn_call:
0x4005bc    mov     $0x6b,%edi
0x4005c1    callq   0x4005b9 <id>
0x4005c6    add     $0x2a,%eax
0x4005c9    retq
```

Function Call Example

%rax	0x95
%rdi	0x6b
%rip	0x4005c9




```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
         id:
0x4005b9   mov    %edi,%eax
0x4005bb   retq
         fn_call:
0x4005bc   mov    $0x6b,%edi
0x4005c1   callq  0x4005b9 <id>
0x4005c6   add    $0x2a,%eax
0x4005c9   retq
```

Function Call Example

%rax	0x95
%rdi	0x6b
%rip	

%rsp → ...

```
int id(int x) { return x; }
int fn_call() { return id(107) + 42; }
```

```
          id:
0x4005b9    mov     %edi,%eax
0x4005bb    retq
          fn_call:
0x4005bc    mov     $0x6b,%edi
0x4005c1    callq   0x4005b9 <id>
0x4005c6    add     $0x2a,%eax
0x4005c9    retq
```

Instructions for Function Calls

`call [addr]`: Jump to the start of a function

- Push `%rip` (return address) onto stack
- Jump to `[addr]`

`ret`: Return from a function

- Pop `%rip` off the stack (effectively jumping)

So Far

Using the stack for local variables

- How the stack works, how to manage it

Function call and return in assembly

- Instructions, calling convention

Using the stack to save registers

Sharing Registers

All functions share same 16 registers

Example:

```
int binky(int x, int y) { return add_5(y) + x; }
```

`binky()` needs x after calling `add_5(y)`

x stored in `%edi` (1st param)

But need to pass y as first param to add_5

Sharing Registers

All functions share same 16 registers

Example:

- `int binky(int x, int y) { return add_5(y) + x; }`
- `binky()` needs x after calling `add_5(y)`
- x stored in `%edi` (1st param)
- But need to pass y as first param to add_5

Solution: Let's save x in `%ebx`

- But wait! What if add_5 uses %ebx?
- ...Let's get back to this

Code: binky

Couple More Instructions

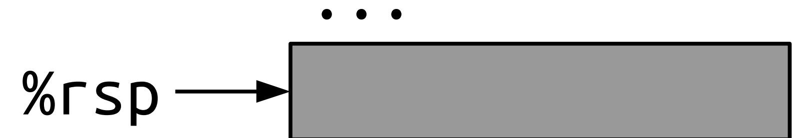
`push [reg]`: Save register on stack

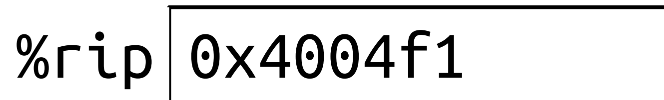
Equivalent to: `sub $8, %rsp; mov [reg],(%rsp)`

`pop [reg]`: Restore register from stack

Equivalent to: `mov (%rsp),[reg]; add $8,%rsp`

Push and Pop

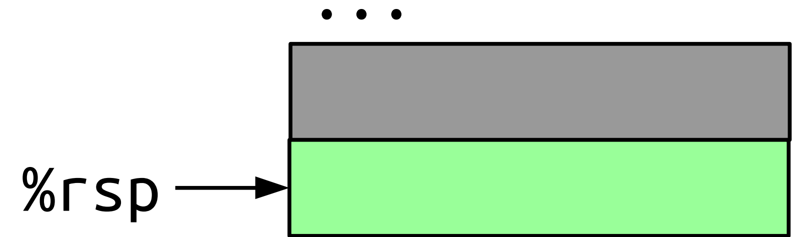





```
int binky(int x, int y)
{ return add_5(y) + x; }
```

```
          binky:
0x4004f1    push    %rbx
            ...
0x4004fd    pop     %rbx
0x4004fe    retq
```

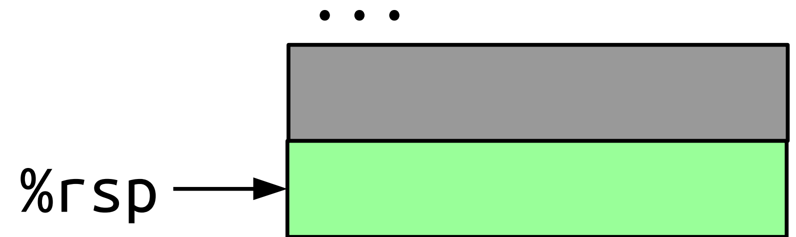
Push and Pop




```
int binky(int x, int y)
{ return add_5(y) + x; }
```

```
          binky:
0x4004f1    push    %rbx
            ...
0x4004fd    pop     %rbx
0x4004fe    retq
```

Push and Pop

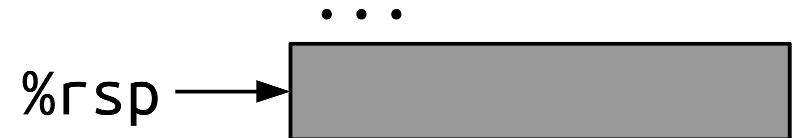


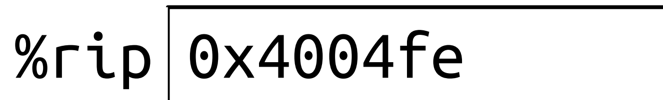
%rip 0x4004fd

```
int binky(int x, int y)
{ return add_5(y) + x; }
```

```
         binky:
0x4004f1   push   %rbx
           ...
0x4004fd   pop    %rbx
0x4004fe   retq
```

Push and Pop






```
int binky(int x, int y)
{ return add_5(y) + x; }
```

```
        binky:
0x4004f1   push    %rbx
           ...
0x4004fd   pop     %rbx
0x4004fe   retq
```

Caller- and callee-saved

Callee-saved register

All functions agree to save old value before use (and restore before return)

Value preserved across function call

`%rbx, %rbp, %r12-%r15`

Caller-saved register

Functions can overwrite without saving

Value not preserved across function call

`%rax, %rdi, %rsi, ...`

Code: winky

Big Picture: The Stack

Use stack when can't use registers

- Too big/too many local variables
- Save values across function calls (return address, callee-saved registers)

Stack makes recursion easy

- Track where we are in recursion
- Preserve needed state during recursive calls

Summary

Using the stack for local variables

How the stack works, how to manage it

Function call and return in assembly

Instructions, calling convention

Using the stack to save registers