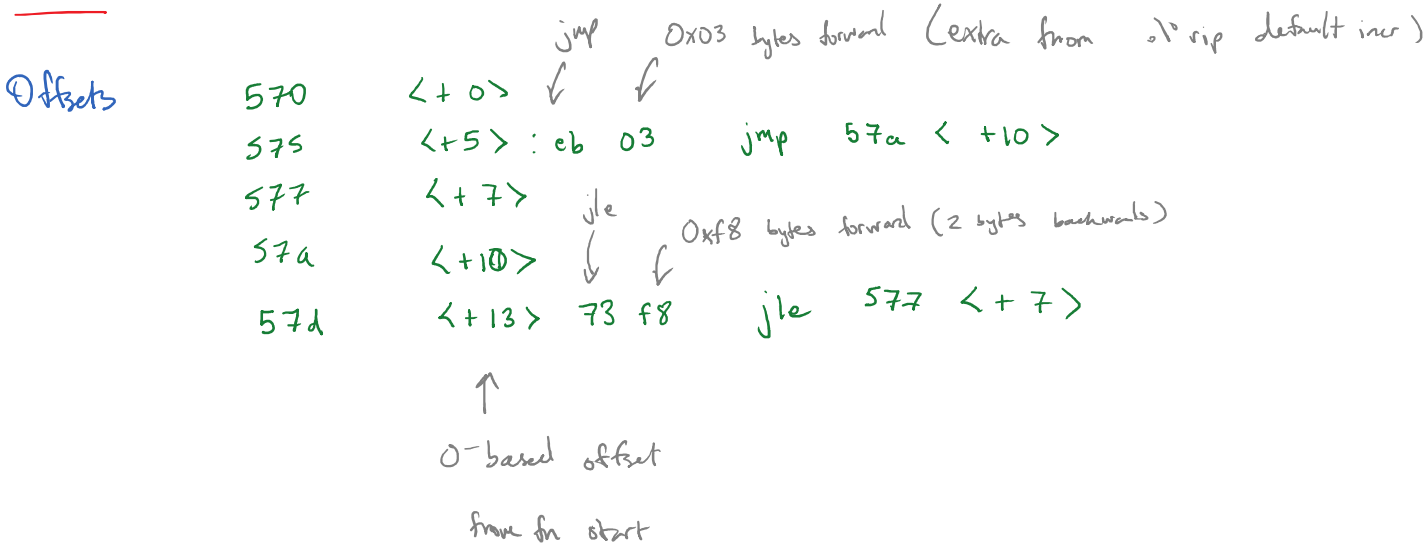
Assembly Function Calls and the Runtime Stack

Monday, October 26, 2020 12:56 PM

%rip



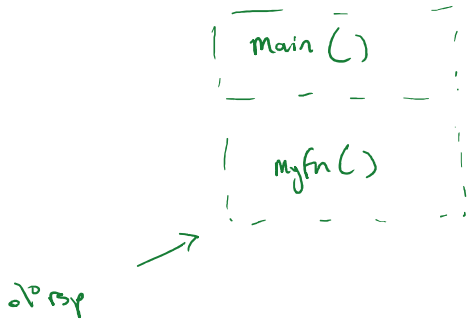
Assembly fns

To call a fn in asm, need to

- → pass control
- → pass data (params, ret val)
- → Manage memory (callee needs on stack frame)

The Stack

%rsp Special register - stores current top of stack



Points to same place before and after fn call

push Pushes data at source onto top of stack, adjusting %rsp accordingly

```
pushq S
```

R[%rsp] ← R[%rsp] - 8
M[R[%rsp]] ← S

OR

```
subq $8, %rsp
movq S, (%rsp)
```

pop Pops data from top of stack, storing in specified destination, adjusting %rsp accordingly

```
popq D
```

D ← M[R[%rsp]]
R[%rsp] ← R[%rsp] + 8

OR

(does not clear data, just increments %rsp to denote that next push can overwrite)

```
movq (%rsp), D
addq $8, %rsp
```

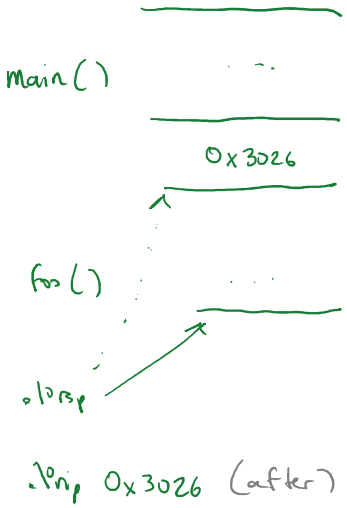
Passing Control

Problem: %rip points to next instruction to execute

To call fn, need to remember next caller instr to resume after

Soln. Push value onto stack, <u>then</u> call fn. %rsp will point to location to restore after fn exits

ex.



call — Pushes instr after call, sets %rip to point to beginning of specified fn

call Label
call *Operand

ret — Pops instruction addr from stack, stores in %rip

Stored %rip value for fn (next instr after exit) is return address (ret value)

Passing Data

Special registers store params, ret vals

In order: %rdi %rsi %rdx %rcx %r8 %r9

Params beyond first 6 are on the stack

Return value goes in %rax

Local Storage

Optimized to put local vars in registers rather than stack

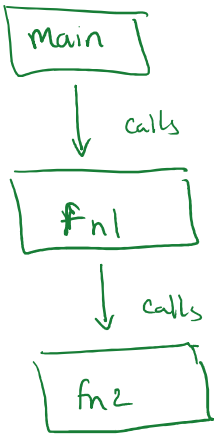
If
- → Run out of registers
- → & operator used (need address)
- → Arrays or structs used (addr arithmetic needed)

Must put data in memory

Register restrictions

→ Only one copy of registers for all programs, functions

→ Define Caller-owned, Callee-owned registers



Rules.

Caller-owned

Callee must save existing value and restore it when done

Caller can store value and assume it will be preserved

Callee-owned

Callee does not need to save existing value

Caller's vals could be overwritten by callee - consider saving elsewhere

Optimizations

`nop` / `nopl` — does nothing, aligns fns on multiples of 8

`mov %ebx, %ebx` — zeroes out top 32 register bits of `%rbx`

`xor %ebx, %ebx` — Optimizes perf, code size (rather than `mov $0x0, %ebx`)

For structure

Initialization

Jump to test
Body
Update
Test
Jump to body if success

$\Rightarrow$

Initialization
Test
Jump past body if fail
Body
Update
Jump to test

repeats are less instruction

Both have the same static instruction count (# written instructions)
but different dynamic instruction count (# executed instructions)

If $n = 0$, right is best
n large, left is best

Conditional move — can eliminate branches (jumps) which are inefficient
(this makes speculative execution easier)