Assembly Language Programming V

x86-64 Architecture

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

OntarioTech UNIVERSITY

Outline

- Creating functions
  - Function definitions
  - Passing arguments
  - Returning values

Defining Functions

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

Stack Instructions

- These instructions are the most basic way to use the stack
  - `push rax` – puts the value of `rax` onto the top of the stack
    - `rsp` is decreased by the size of `rax` (8)
    - `rsp` points to the top of the stack
  - `pop rax` – pops the top of the stack into `rax`
    - `rsp` is increased by the size of `rax` (8)

Note: The stack in x64 (and x86) grows downward
i.e. the stack grows toward lower addresses, not higher addresses

The Calling Stack

- Every program has a dedicated program stack
- Each time a function is called:
  - Some arguments may be passed via registers
  - Other arguments may be pushed onto the stack
    - These arguments are pushed in reverse order
  - The return address (`rip`) is saved onto the stack
  - The stack base pointer (`rbp`) is saved onto the stack

Passing Arguments via Registers

- System V AMD64 ABI (application binary interface):
  - A calling convention used by Linux, MacOS, BSD
  - For integer or pointer arguments:
    - `rdi` - first integer/pointer argument
    - `rsi` - second integer/pointer argument
    - `rdx` - third integer/pointer argument
    - `rcx` - fourth integer/pointer argument
    - `r8` - fifth integer/pointer argument
    - `r9` - sixth integer/pointer argument
  - All remaining arguments are passed via the stack

https://web.archive.org/web/20160801075139/http://www.x86-64.org/documentation/abi.pdf

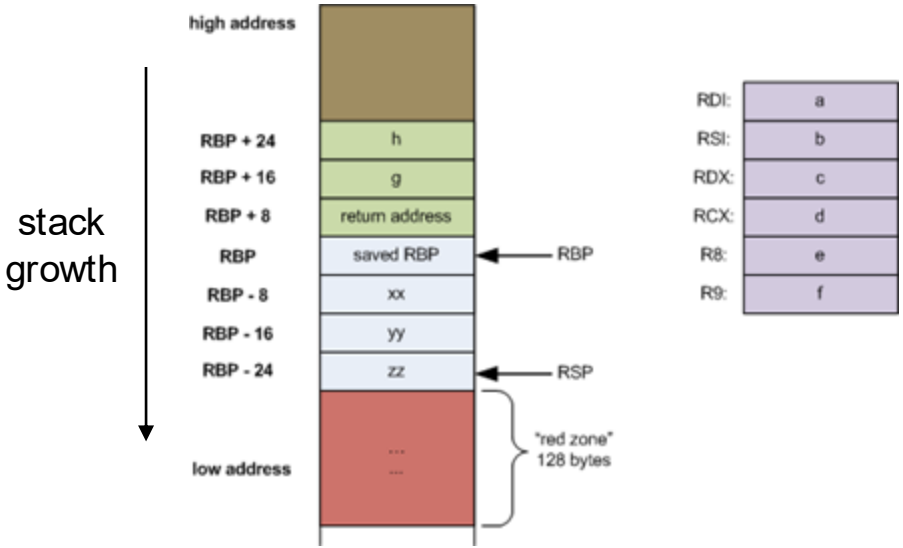
Passing Arguments via the Stack

- System V AMD64 ABI:
  - Recall that rsp points to the top of the stack
  - When calling a function
    - `[rsp]` contains the return address
    - `[rsp+8]` contains the first stack parameter

https://web.archive.org/web/20160801075139/http://www.x86-64.org/documentation/abi.pdf

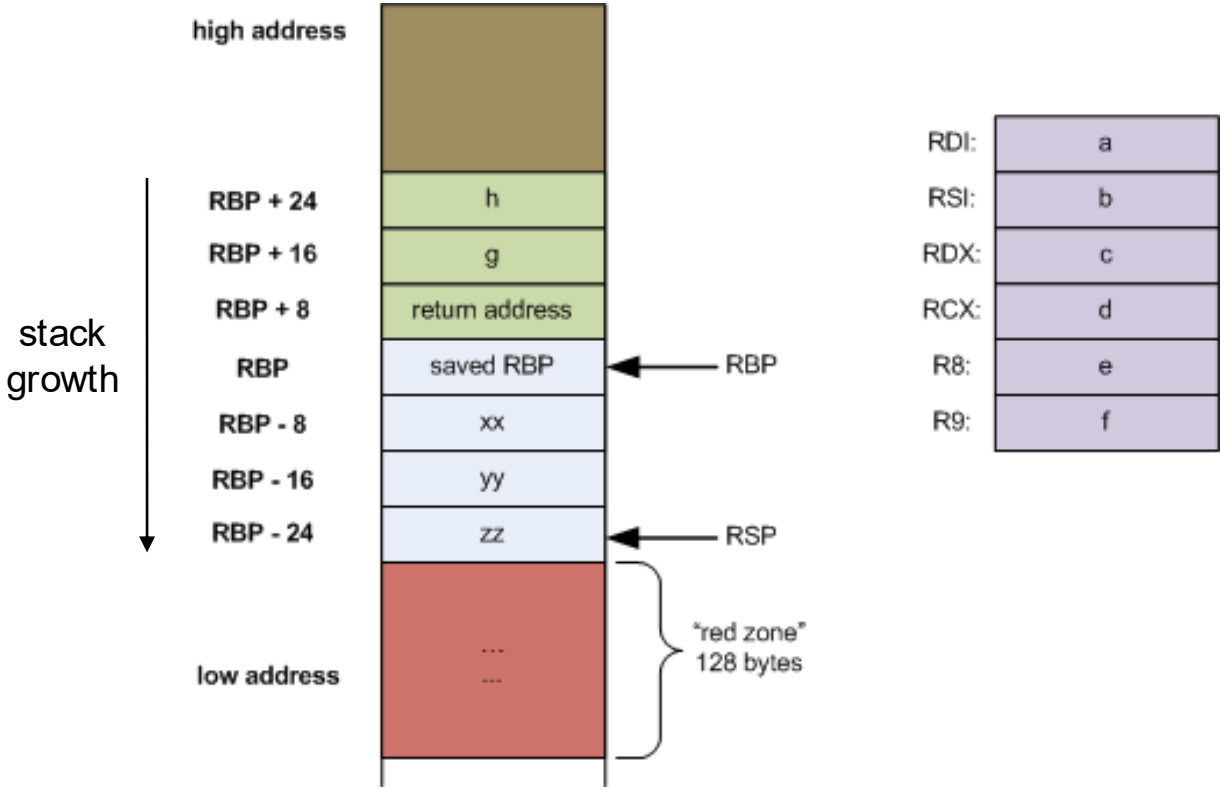
The Calling Stack

```
long func1(long a, long b, long c, long d,
           long e, long f, long g, long h) {
    long xx, yy, zz;
    ...
}
```



https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

The Calling Stack



https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

Returning from a Function

- A function returns using the `ret` instruction
  - `ret` pops `rbp` and `rip` off the stack, and resumes execution at `rip + (instruction_size)`
  - If there are still local variables on the stack, these should be popped off
    - If you do not, your program will almost certainly crash
  - If there are still arguments on the stack, you can't pop them off
    - The `rbp` and `rip` values are in the way

Cleaning up the Stack

- A function returns using the `ret` instruction
  - If there are still arguments on the stack, you can't pop them off
    - The `rbp` and `rip` values are in the way
  - There is a version of `ret` that takes an integer operand
    - The number of bytes to pop off the stack after popping `rbp`, `rip`
    - e.g. `ret 8` (pop 1 64-bit value off the stack)
  - Alternatively, the caller of the function could pop the values off the stack after the function returns

Wrap Up

- Creating functions
  - Function definitions
  - Passing arguments
  - Returning values

What is next?

- Optimizations
  - Pipelining
  - Hyperthreading