# Function

COP-3402 Systems Software
Paul Gazzillo

# Functions Abstract Away Computation

- Example: keys and locks
  - Input: key
  - Ouput: unlocking
  - Implementation: tumbler, wafer, RFID, etc
- Example: printf
  - Input: format string, values
  - Output: characters to stdout and num chars written
  - Implementation:
    - https://sourceware.org/git/?p=glibc.git;a=blob;f=stdio-common/vfprintf-internal.c;h=547a3a868b4668bf615cf3f39a92e3c11cbb98ad;hb=HEAD#l1288

# Functions Abstract Away Computation

- Abstraction
  - Name the computation
  - Define inputs/outputs
- Reuseable
  - Reason about specification instead of implementation
- Create functions with care
  - Explicitly state assumptions
  - Cover all possible inputs
  - Document any side effects (or just avoid them)

# Functions for Recursive Descent

- Carefully define each parsing function
  - Inputs/outputs
  - Position of file pointer (fgetc, fungetc)
- Example
  - Callee assumes file pointer on preceding
  - Callee leaves leave file pointer on its last character
- Ensure all parsing functions follow the same rules
  - This make the functions composeable
- Easier to just reason about one function at-a-time

# Functions in SimpleC

- Similar to C
- A function has
  - A name, e.g., square
  - Parameter types, e.g., int
  - A return type, e.g., int
- Parameters are inputs
- Return creates the output
- Side effects
  - Global values may be (implicit) inputs and outputs

```
int square(int x) {
  return x * x;
}
```

square : (int) -> int

# Static Scoping

- Region of code where variable is valid
  - Compound statement
  - Nested scopes, e.g., int x
- Local variables
  - Declared in nested scopes
  - Function parameters
- Out-of-scope variables
  - e.g., print y

```
int x;
int f(int x) {
  int y;
  read y;
  return x * y;
}
print x;
print y;
```

# Function Application

- (Conceptually) replace function its result

```
print square(9);

print 9 * 9;

print 81;
```

# Simulating Functions in Hardware

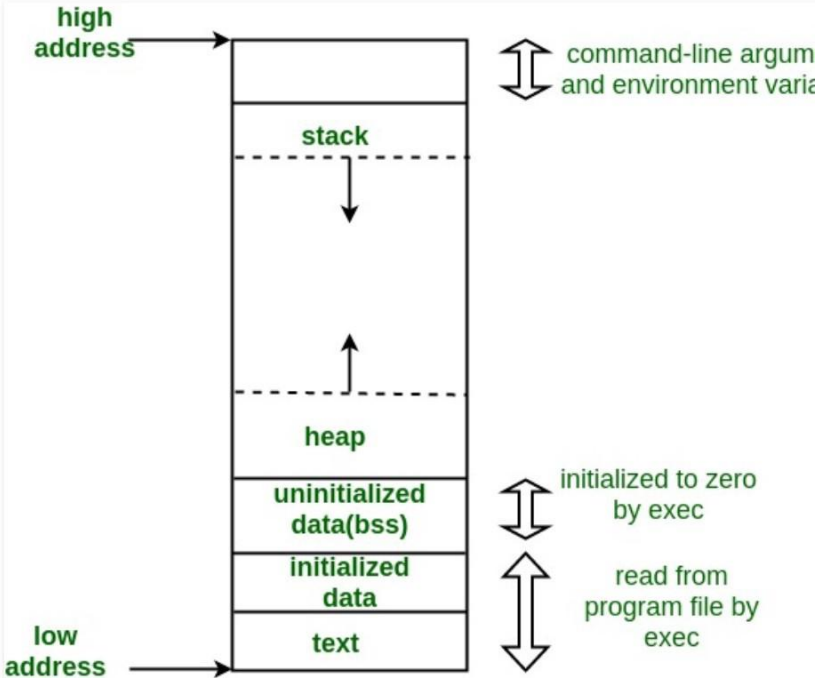
- No real function abstraction in hardware, e.g., x86
- How can we generate code for a function using only
  - Arithmetic and logical operators
  - Memory operations
  - Compare and branch
- Answer
  - Branch to code of function (or inline it)
  - Copy parameters to well-known registers & memory locations
  - Copy return value to well-known register or memory location

# Demo: Functions in Assembly

- Functions as unconditional branches
- Passing parameters
- Local state
- Return value

# Freezing Function State Using the Stack

- Stack holds local variables






https://www.linux.com/tutorials/stack-vs-heap-whats-difference-and-why-should-i-care/

# Demo: Freezing Function State on the Stack



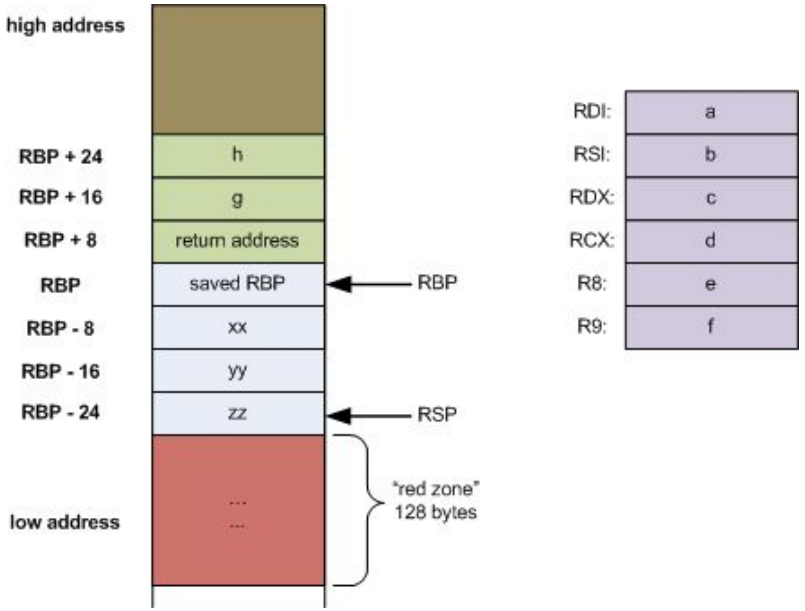
https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/

# Functions in Intel Assembly

- Calling conventions
  - Standard for caller/callee communication
- Application Binary Interface (ABI)
- Linux uses System V AMD64 ABI
- Parameters passed in six specific registers
  - rdi, rsi, etc
  - More than six, more parameters in stack
- Return value in specific register (eax)

# System V AMD64 ABI

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```




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

# Demo: Intel Assembly