Investigating the Code

8

CS/COE 0449
Introduction to
Systems Software

wilkie
(with content borrowed from Vinicius Petrucci
and Jarrett Billingsley)

Spring 2019/2020

Going with the Flow

Tracing the footsteps

Bringing back our alphabet soup: The C ABI

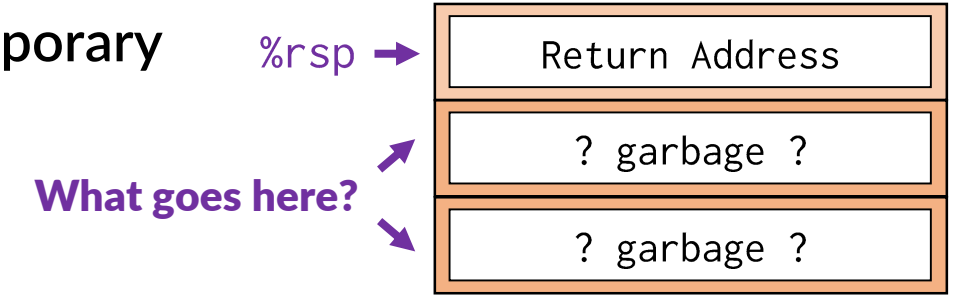
- The **C Application Binary Interface** (ABI) are assembly conventions
  - Like MIPS, certain registers are typically used for returns values, args, etc
  - It is not defined by the language, but rather the OS.
    - Windows and Linux (UNIX/System V) have a different C ABI ☹

- In our x86-64 Linux C ABI, registers are used to pass arguments:
  - `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (First, second, etc) (Like MIPS `a0` – `a3`)
  - Remaining arguments go on the stack.
  - Callee must preserve `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, `%r15` (Like MIPS `s0` – `s7`)
  - Return value: `%rax` (overflows into `%rdx` for 128-bits) (MIPS `v0` – `v1`)
  - Lots of other small things not worth going over.

- For reference: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf

Function, function… what's your… function

- The **activation frame** contains temporary data needed by the function.
  - `%rax` is the return value
  - `%rsp` is the current stack address
  - `%rbp` ***is the address of this frame***



C

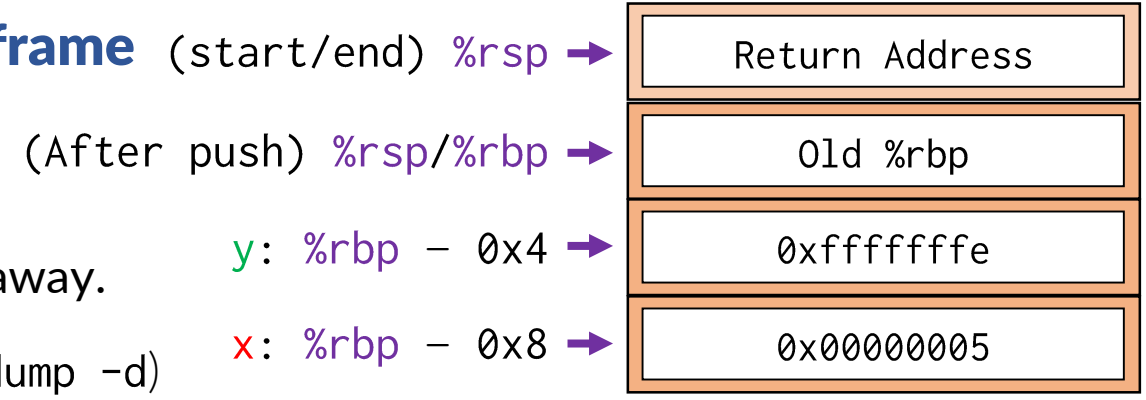
```c
int main(void) {
  int x = 5;
  int y = -2;
  if (x < 0) {
    x = -x;
  }
  if (x < y) {
    x = y;
  }
  return 0;
}
```

x86-64 (gas / AT&T syntax, `objdump -d`)

```
0000000000001119 <main>:
    1119:       55                      push   %rbp
    111a:       48 89 e5                mov    %rsp,%rbp
    111d:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
    1124:       c7 45 fc fe ff ff ff    movl   $0xfffffffe,-0x4(%rbp)
    112b:       83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
    112f:       79 03                   jns    1134 <main+0x1b>
    1131:       f7 5d f8                negl   -0x8(%rbp)
    1134:       8b 45 f8                mov    -0x8(%rbp),%eax
    1137:       3b 45 fc                cmp    -0x4(%rbp),%eax
    113a:       7d 06                   jge    1142 <main+0x29>
    113c:       8b 45 fc                mov    -0x4(%rbp),%eax
    113f:       89 45 f8                mov    %eax,-0x8(%rbp)
    1142:       b8 00 00 00 00          mov    $0x0,%eax
    1147:       5d                      pop    %rbp
    1148:       c3                      retq
```

Oh, that's your function

- **First: it fills the activation frame with initial variable values.**
  - It may not allocate them in any strict order. Here, it allocates x first *and* further away.



x86-64 (gas / AT&T syntax, `objdump -d`)

```
0000000000001119 <main>:
    1119:       55                      push   %rbp
    111a:       48 89 e5                mov    %rsp,%rbp
    111d:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
    1124:       c7 45 fc fe ff ff ff    movl   $0xfffffffe,-0x4(%rbp)
    112b:       83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
    112f:       79 03                   jns    1134 <main+0x1b>
    1131:       f7 5d f8                negl   -0x8(%rbp)
    1134:       8b 45 f8                mov    -0x8(%rbp),%eax
    1137:       3b 45 fc                cmp    -0x4(%rbp),%eax
    113a:       7d 06                   jge    1142 <main+0x29>
    113c:       8b 45 fc                mov    -0x4(%rbp),%eax
    113f:       89 45 f8                mov    %eax,-0x8(%rbp)
    1142:       b8 00 00 00 00          mov    $0x0,%eax
    1147:       5d                      pop    %rbp
    1148:       c3                      retq
```

← **Preserves `%rbp` (caller activation frame)**

Allocates "x" on stack (`-8` from top)

Allocates "y" on stack (`-4` from top)

(it does not have to be in order)

← **Resets caller activation frame**

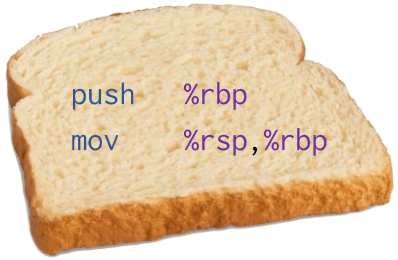
← **Returns (return value is in `%rax`)**

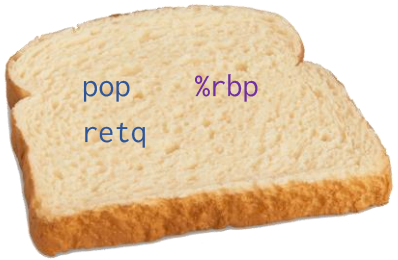
These are actual sandwiches (no hot dogs or w/e)

```
0000000000001119 <main>:
    1119:       55                      push   %rbp
    111a:       48 89 e5                mov    %rsp,%rbp

    111d:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
    1124:       c7 45 fc fe ff ff ff    movl   $0xfffffffe,-0x4(%rbp)
    112b:       83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
    112f:       79 03                   jns    1134 <main+0x1b>
    1131:       f7 5d f8                negl   -0x8(%rbp)
    1134:       8b 45 f8                mov    -0x8(%rbp),%eax
    1137:       3b 45 fc                cmp    -0x4(%rbp),%eax
    113a:       7d 06                   jge    1142 <main+0x29>
    113c:       8b 45 fc                mov    -0x4(%rbp),%eax
    113f:       89 45 f8                mov    %eax,-0x8(%rbp)
    1142:       b8 00 00 00 00          mov    $0x0,%eax

    1147:       5d                      pop    %rbp
    1148:       c3                      retq
```





- When identifying functions, you are looking for that tell-tale sandwich pattern.
- A `push` is a good sign of the beginning of a function
- And the `pop` will happen before the `ret` at the end.
- Everything between is the sweet, sweet jam that makes it unique.

Who controls the `cmp` controls the flow

- Control flow is a `cmp` or `test` followed by `j*`
  - `cmp` will set `FLAGS` based on the difference (subtraction) between values
  - `test` will set `FLAGS` based on bitwise AND of both values (faster, but less useful)
- `j*` group set `%rip` (program counter) to an address based on `FLAGS`
  - Often it is much more useful to just interpret the `jmp` (`jge` is `>=`)

C

```
if (x < 0) {
  x = -x;
}
```

```
if (x < y) {
  x = y;
}
```

x86-64 (gas / AT&T syntax, `objdump -d`)

```
cmpl   $0x0,-0x8(%rbp)
jns    1134 <main+0x1b>
negl   -0x8(%rbp)

mov    -0x8(%rbp),%eax
cmp    -0x4(%rbp),%eax
jge    1142 <main+0x29>
mov    -0x4(%rbp),%eax
mov    %eax,-0x8(%rbp)
```

Who controls the `cmp` controls the flow

- `FLAGS` has bits that are set based on the ALU (CPU math logic) result
  - `SF` – most significant bit of result
  - `OF` – set if overflow occurred
  - `ZF` – set if result is zero
  - `CF` – set if last bit operation has carry
- Each jump looks at different `FLAGS` patterns. (Look 'em up!)
  - `jns` – Jumps when `SF = 0`
  - `jge` – set if `SF = OF` or `ZF = 1`

C

```
if (x < 0) {
  x = -x;
}
```

Works because of 2's complement math. (thus, instead of its strict definition, better to think about it abstractly)

```
if (x < y) {
  x = y;
}
```

x86-64 (gas / AT&T syntax, `objdump -d`)

```
cmpl   $0x0,-0x8(%rbp)
jns    1134 <main+0x1b>
negl   -0x8(%rbp)

mov    -0x8(%rbp),%eax
cmp    -0x4(%rbp),%eax
jge    1142 <main+0x29>
mov    -0x4(%rbp),%eax
mov    %eax,-0x8(%rbp)
```

← Perform x - 0 (does nothing!)

← Jump if the result (that is, x) does not have a set sign bit. (x is positive in that case)

← Perform x - y

← Jump if the result is 0 or if result is negative after overflow or positive and didn't overflow. (x is >= y in these cases)

cmp, simplifying… the confusion

- Just remember that the order of operands is not the… best order…
  - It's kinda swapped around in the AT&T syntax we have been looking at:

```
if (x < 0) {
}
```

```
cmpl    0, x
jns     <address>
```
← Jump if x > 0

```
if (x < y) {
}
```

```
cmpl    y, x
jge     <address>
```
← Jump if x >= y

```
if (x >= y) {
}
```
← We negate the condition →

```
cmpl    y, x
jl      <address>
```
← Jump if x < y

```
if (x == y) {
}
```
Because we are deciding when to skip the code!

```
cmpl    y, x
jne     <address>
```
← Jump if x != y

test… adding some new confusion

- `test` is somewhat stranger… and requires some more thought.
  - performs an AND of the arguments and sets flags on result

- Thankfully, generally only commonly used in a couple of cases.
  - Generally to test a value against "true" or "false".
  - Recall that `jne` and `je` will look at the zero flag (`FLAGS[ZF]`)
  - Keep in mind that jumps are built around `cmp` (which performs: `x - x`)…

```
if (!x) {
}
```

← We negate the condition →

```
test    x, x
jne     <address>
```

← Jump if x != 0 (ZF = 0?)

```
if (x) {
}
```

```
test    x, x
je      <address>
```

← Jump if x == 0 (ZF = 1?)

Patterns

- Control flow is a `cmp` / `test` before a `j*`

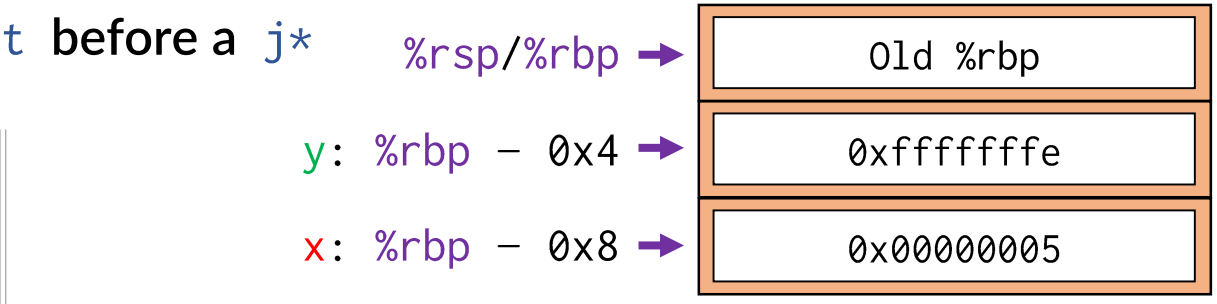
C

```
int main(void) {
  int x = 5;
  int y = -2;

  if (x < 0) {
    x = -x;
  }

  if (x < y) {
    x = y;
  }

  return 0;
}
```



x86-64 (gas / AT&T syntax, `objdump -d`)

```
0000000000001119 <main>:
    1119:       55                      push   %rbp
    111a:       48 89 e5                mov    %rsp,%rbp
    111d:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
    1124:       c7 45 fc fe ff ff ff    movl   $0xfffffffe,-0x4(%rbp)
    112b:       83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
    112f:       79 03                   jns    1134 <main+0x1b>
    1131:       f7 5d f8                negl   -0x8(%rbp)
    1134:       8b 45 f8                mov    -0x8(%rbp),%eax
    1137:       3b 45 fc                cmp    -0x4(%rbp),%eax
    113a:       7d 06                   jge    1142 <main+0x29>
    113c:       8b 45 fc                mov    -0x4(%rbp),%eax
    113f:       89 45 f8                mov    %eax,-0x8(%rbp)
    1142:       b8 00 00 00 00          mov    $0x0,%eax
    1147:       5d                      pop    %rbp
    1148:       c3                      retq
```

Altogether now… Working backward

```
0000000000001119 <main>:
    1119: push   %rbp                    // Preserve caller's %rbp on stack
    111a: mov    %rsp,%rbp               // Set %rbp to %rsp
    111d: movl   $0x5,-0x8(%rbp)         // Store x on stack with value of 5
    1124: movl   $0xfffffffe,-0x4(%rbp)  // Store y on stack with value of -2
    112b: cmpl   $0x0,-0x8(%rbp)
    112f: jns    1134 <main+0x1b>        // if (x < 0) {   ← Negate logic to form "if" logic
    1131: negl   -0x8(%rbp)              //   x = -x
                                         // }
    1134: mov    -0x8(%rbp),%eax
    1137: cmp    -0x4(%rbp),%eax
    113a: jge    1142 <main+0x29>        // if (x < y) {   ← Negate logic to form "if" logic
    113c: mov    -0x4(%rbp),%eax
    113f: mov    %eax,-0x8(%rbp)         //   x = y
                                         // }
    1142: mov    $0x0,%eax
    1147: pop    %rbp                    // Recall caller's %rbp from stack
    1148: retq                           // return 0
```

Deduction, dear watson

```
0000000000001119 <main>:
    1119: push   %rbp
    111a: mov    %rsp,%rbp
    111d: movl   $0x5,-0x8(%rbp)
    1124: movl   $0xfffffffe,-0x4(%rbp)
    112b: cmpl   $0x0,-0x8(%rbp)
    112f: jns    1134 <main+0x1b>
    1131: negl   -0x8(%rbp)

    1134: mov    -0x8(%rbp),%eax
    1137: cmp    -0x4(%rbp),%eax
    113a: jge    1142 <main+0x29>
    113c: mov    -0x4(%rbp),%eax
    113f: mov    %eax,-0x8(%rbp)

    1142: mov    $0x0,%eax
    1147: pop    %rbp
    1148: retq
```

← **No use of `%rdi` ... likely no arguments**

← **Two stack allocations ... Two local variables. (initialized to 5 and, likely, -2)**

← **Looking at `%rax` ... This simply returns zero.**

Conventional wisdom: counting arguments

```
000000000000112d <main>:
    112d: 55                 push   %rbp
    112e: 48 89 e5           mov    %rsp,%rbp
    1131: be 03 00 00 00     mov    $0x3,%esi
    1136: bf 05 00 00 00     mov    $0x5,%edi
    113b: e8 d9 ff ff ff     callq  1119 <whoknows>
    1140: b8 00 00 00 00     mov    $0x0,%eax
    1145: 5d                 pop    %rbp
    1146: c3                 retq
```

← **Readies `%rsi` ... second argument!**

← **Readies `%rdi` ... first argument!**

Since they are `%e*` ... yep! Both 32-bit!

↖ **Like a `jal` in MIPS. A function call.**

```
???  whoknows ( int a, int b ) {   // Let's assume int is 32 bits

     ???
}
```

← **Still have to follow the `call` to the assembly of the function.**

Conventional wisdom: counting arguments

```
0000000000001119 <whoknows>:
    1119: 55          push   %rbp
    111a: 48 89 e5    mov    %rsp,%rbp
    111d: 89 7d fc    mov    %edi,-0x4(%rbp)
    1120: 89 75 f8    mov    %esi,-0x8(%rbp)
    1123: 8b 55 fc    mov    -0x4(%rbp),%edx
    1126: 8b 45 f8    mov    -0x8(%rbp),%eax
    1129: 01 d0       add    %edx,%eax
    112b: 5d          pop    %rbp
    112c: c3          retq
```

← **Copies** `%rdi` **... function argument!**

← **Copies** `%rsi` **... second argument!**

Since they are `%e*` **... They are both 32-bit!**

← `%rax` **is the return address...**

`%eax` **means it is a 32-bit return**

```
%eax = %eax + %edx
     = a + b
```

```
int  whoknows ( int a, int b ) {   // Let's assume int is 32 bits

     return  a + b  ;

}
```