Some Differences Between x86 and x86-64

December 1, 2016

Big Differences

- Pointers, longs 64 bits
  - can still do 8-, 16-, and 32-bit ops
- 16 general purpose registers ($2x$ as many)
  - $>$ registers $\rightarrow$ less need for stack
  - many variables will fit in registers
  - usually pass function arguments through registers

Procedure Call

arguments

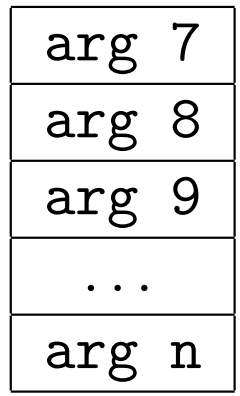
- `%rdi`
- `%rsi`
- `%rdx`
- `%rcx`
- `%r8`
- `%r9`

return value

- `%rax`

stack

only when $>$ 6 arguments



Register Saving Convention

caller saved

- RAX
- RCX
- RDX
- R8
- R9
- R10
- R11

callie saved

- RBX
- RBP
- RDI
- RSI
- RSP
- R12
- R13
- R14
- R15

A Simple Example

C

```
int func(int x, int y, int z) {
  return x+y+z;
}
```

x86

```
func:
        movl        8(%esp), %eax
        addl        4(%esp), %eax
        addl        12(%esp), %eax
        ret
```

x86-64

```
func:
        leal        (%rdi,%rsi), %eax
        addl        %edx, %eax
        ret
```

A Simple Main

C

```
int main(int argc, char **argv) {
  int a=10, b=20, c=30;
  func(a, b, c);
  return EXIT_SUCCESS;
}
```

x86-64

```
main:
 ...

 movl $30, %edx
 movl $20, %esi
 movl $10, %edi
 call func

 ...
```

x86

```
main:
  ...

  movl    $10, -20(%ebp)
  movl    $20, -16(%ebp)
  movl    $30, -12(%ebp)
  subl    $4, %esp
  pushl   -12(%ebp)
  pushl   -16(%ebp)
  pushl   -20(%ebp)
  call    func
  addl    $16, %esp

  ...
```

Why %eax and not %rax, etc.?

C

```
int func(int x, int y, int z) {
  return x+y+z;
}
```

x86-64

```
func:
        leal        (%rdi,%rsi), %eax
        addl        %edx, %eax
        ret
```

answer

- we're using `int`s
- `int`s are 32-bit quantities with this compiler
- %eax is the low order 32 bits of %rax

Same thing with `long`

C

```
long func(long x, long y, long z) {
  return x+y+z;
}
```

x86-64

```
func:
  leaq          (%rdi,%rsi), %rax
  addq          %rdx, %rax
  ret
```