Machine Programming 3: Procedures

CS61, Lecture 5
Prof. Stephen Chong
September 15, 2011
Announcements

• Assignment 2 (Binary bomb) due next week
  • If you haven’t yet please create a VM to make sure the infrastructure works for you
Today

• Procedures
  • The stack
  • Stack frames
  • Leave
  • Register conventions
• x86_64
• How do we call procedures?
• Where do we store local variables (e.g., x,y,z)?
• How do we return values from procedures?
• How do we support recursion?
Stack

• Stack is used for handling function calls and local storage
  • Stores local variables, return address, saved registers, ...
• Stack pointer %esp always holds address of top stack element
• Stack grows downwards!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing addresses

0x108

Stack "top"
Pushing and popping

- Two data movement instructions for stack: `pushl` and `popl`
  - `pushl src`
    - Push four bytes onto stack
    - Effect is
      
      \[
      R[\%esp] \leftarrow R[\%esp] - 4 \\
      M[R[\%esp]] \leftarrow src
      \]
  - E.g., `pushl %eax`

\[ %\text{eax} \quad 0\times123 \\
    %\text{edx} \quad 0 \\
    %\text{esp} \quad 0\times108 \]

Stack "bottom"

Increasing addresses

0x108

Stack "top"
Pushing and popping

- Two data movement instructions for stack: `pushl` and `popl`

  - `pushl src`
    - Push four bytes onto stack
    - Effect is
      \[
      R[\%esp] \leftarrow R[\%esp] - 4
      \]
      \[
      M[R[\%esp]] \leftarrow src
      \]
  - E.g., `pushl %eax`
Pushing and popping

- `popl dest`
  - Pops four bytes from stack
  - Effect is
    \[
    dest \leftarrow M[R[\%esp]]
    \]
    \[
    R[\%esp] \leftarrow R[\%esp] + 4
    \]
  - E.g., `popl %edx`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing addresses

0x108
0x104

0x123

Stack "top"
Pushing and popping

- `popl dest`
  - Pops four bytes from stack
  - Effect is
    \[ \text{dest} \leftarrow M[R[\%esp]] \]
    \[ R[\%esp] \leftarrow R[\%esp] + 4 \]
  - E.g., `popl %edx`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0x123</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"
Examining the stack

- Can use `movl` to access and modify arbitrary values on the stack
  - No need to access just top element
  - Can “peek” at stack:
    - `movl 12(%esp), %eax`
  - Can “poke” stack:
    - `movl $0xdeadbeef, 12(%esp)`

```
|   %eax   | 0x123 |
|   %edx   | 0x123 |
|   %esp   | 0x108 |
```

Stack “bottom”

Stack “top”

Increasing addresses
Procedure control flow

- Stack is used to implement procedure call and return

- Procedure call
  - x86 instruction: `call address`
  - Pushes `return address` on stack, then jumps to `address`
  - What is the return address?
    - Address of instruction after the `call` instruction
    - E.g.,
      - Return address is 0x8048553

- Procedure return
  - x86 instruction: `ret`
  - Pops return address from stack, and jumps to it
Procedure call example

Before call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>0x123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x0108</td>
</tr>
<tr>
<td>%eip</td>
<td>0x804854e</td>
</tr>
</tbody>
</table>

After call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x0104</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8048b90</td>
</tr>
</tbody>
</table>
Procedure call example

Before call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>0x123</td>
</tr>
</tbody>
</table>

| %esp    | 0x108  |
| %eip    | 0x804854e |

After call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>0x123</td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

| %esp    | 0x104  |
| %eip    | 0x8048b90 |

804854e: e8 3d 06 00 00    call 8048b90 <main>
8048553: 50             pushl %eax
Procedure return example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x114</td>
<td>0x104</td>
</tr>
<tr>
<td>0x110</td>
<td>0x123</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x804853</td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048591</td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
</tbody>
</table>

Before return

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8048b91</td>
</tr>
</tbody>
</table>

After return

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%eip</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>
Stack-based languages

• Languages that support recursion
  • E.g., C, Pascal, Java
  • Must be able to support multiple instantiations of single procedure
    • Code must be **reenterant**

• Each invocation of a procedure has its own local state
  • Arguments to the procedure (e.g., `x`)
  • Local variables within the procedure (e.g., `rval`)
  • Return address

• Where are these stored?

```c
int rfact(int x) {
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```
Each procedure invocation has an associated **stack frame**
- The “chunk” of the stack for that procedure invocation
- Contains local variables, arguments to functions, and return address
- Needed from when procedure called to when it returns

**Stack discipline**
- Stack frame released when procedure returns
- Callee must return before caller does

**Current stack frame described by two registers**
- **%ebp**: frame pointer
  - Points to base (or “bottom”) of current stack frame
- **%esp**: stack pointer
  - Points to stop of stack (i.e., top of current stack frame)
Stack frame example

```c
void foo(...) {
    ...
    bar();
    ...
}

void bar(...) {
    int x, y;
    x = baz();
    ...
    y = quux();
    ...
}

int baz(...) {
    int z;
    ...
    z = baz();
    ...
    return z;
}

int quux(...) {
    ...
    return 42;
}
```

Call chain:
- `foo` → `bar` → `baz` → `baz` → `baz` → `quux`
Stack frame example

Call chain

```c
void foo(...) {
    ...
    bar();
    ...
}
```

%esp ───> %ebp ───> foo
Stack frame example

void foo(...) {
...
bar();
...
}

void bar(...) {
  int x, y;
  x = baz();
  ...
  y = quux();
  ...
}
void foo(...) {
...
bar();
...
}

void bar(...) {
int x, y;
x = baz();
...
y = baz();
...
}

int baz(...) {
int z;
...
z = baz();
...
return z;
}
void foo(...) {
...
bar();
...
}

void bar(...) {
    int x, y;
    x = baz();
    ...
    y = baz();
    ...
}

int baz(...) {
    int z;
    ...
    z = baz();
    ...
    return z;
}
Stack frame example

void foo(...) {
...
bar();
...
}

void bar(...) {
int x, y;
x = baz();
...
y = baz();
...
}

int baz(...) {
int z;
...
z = baz();
...
return z;
}

Stephen Chong, Harvard University
void foo(...) {
  ...
  bar();
  ...
}

void bar(...) {
  int x, y;
  x = baz();
  ...
  y = baz();
  ...
}

int baz(...) {
  int z;
  ...
  z = baz();
  ...
  return z;
}

int baz(...) {
  int z;
  ...
  z = baz();
  ...
  return z;
}
Stack frame example

Call chain

```c
void foo(...) {
    ...
    bar();
    ...
}

void bar(...) {
    int x, y;
    x = baz();
    ...
    y = baz();
    ...
}

int baz(...) {
    int z;
    ...
    z = baz();
    ...
    return z;
}
```
Stack frame example

```c
void foo(...) {
  ...
  bar();
  ...
}

void bar(...) {
  int x, y;
  x = baz();
  ...
  y = quux();
  ...
}
```

Call chain

- `foo`
- `bar`
- `baz`
Stack frame example

```c
void foo(...) {
    ...
    bar();
    ...
}

void bar(...) {
    int x, y;
    x = baz();
    ...
    y = baz();
    ...
}

int quux(...) {
    ...
    return 42;
}
```

Call chain:
- `foo`
- `bar`
- `baz`
- `quux`

Stack frame:
- `%ebp`:
  - `foo`
- `%esp`:
  - `bar`
  - `quux`
Stack frame example

```c
void foo(...) {
    ...
}

void bar(...) {
    int x, y;
    x = baz();
    ...
    y = quux();
    ...
}

void bar(...) {
    ...
    y = quux();
    ...
}
```

Call chain:
- foo
  - bar
    - baz
      - baz
      - baz
      - quux
    - quux
void foo(...) {
   ...
   bar();
   ...
}

Call chain:
- foo
  - bar
    - baz
      - baz
        - baz
          - baz

%ebp → %esp → foo
x86/Linux stack frame

- The exact layout of a stack frame is a convention.
  - Depends on hardware, OS, and compiler used.

- x86/Linux stack frame contains:
  - Old value of `%ebp` (from previous frame)
  - Any saved registers (more later)
  - Local variables (if not kept in registers)
  - Arguments to function about to be called

- The **caller's** stack frame contains:
  - Return address – pushed by call instruction
  - Arguments for this function call
swap revisited

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```c
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

void call_swap() {
    swap(&zip1, &zip2);
}
```

call_swap:

```assembly
...  
pushl $zip2    # Push args
pushl $zip1    # on stack
call swap      # Do the call
...  
```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

void call_swap() {
    swap(&zip1, &zip2);
}

/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

call_swap:
    ...
    pushl $zip2    # Push args
    pushl $zip1    # on stack
    call swap      # Do the call
    ...

Stack
%ebp
%esp
...
Swap revisited

/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

void call_swap() {
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

call_swap:
    ...
    pushl $zip2      # Push args
    pushl $zip1      # on stack
    call swap        # Do the call
    ...

Stack
%ebp ➔ ...
%esp ➔ $zip2
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

void call_swap() {
    swap(&zip1, &zip2);
}

call_swap:
    ...
    pushl $zip2        # Push args
    pushl $zip1        # on stack
    call swap          # Do the call
    ...

Stack

%ebp ➔

...$zip2

%esp ➔

$zip1
Swap revisited

```c
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

void call_swap() {
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

call_swap:
...
pushl $zip2  # Push args
pushl $zip1  # on stack
call swap   # Do the call
...

Stack

%ebp  
...
$zip2
$zip1
Return address

%esp  
Stephen Chong, Harvard University
Code for swap

```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Swap setup

Stack entering swap

%ebp →

...  
$zip2  
$zip1  

%esp → Return address

Resulting stack

%ebp →

...  
$zip2  
$zip1  

%esp → Return address

pushl %ebp
movl %esp,%ebp
pushl %ebx

Set up
Swap setup

Stack entering swap

%ebp →
...
$zip2
$zip1
%esp → Return address

Resulting stack

%ebp →
...
$zip2
$zip1
%esp → Return address
%esp → Old %ebp

pushl %ebp
movl %esp,%ebp
pushl %ebx

Set up
Swap setup

Stack entering swap

%ebp → ...

$zip2

$zip1

%esp → Return address

Resulting stack

%ebp → ...

$zip2

$zip1

%esp → Return address

%ebp → Old %ebp

Set up

pushl %ebp
movl %esp,%ebp
pushl %ebx
Swap setup

Stack entering swap

%ebp → ...
%esp → Return address

$zip2
$zip1

Resulting stack

...$zip2$zip1
Return address
Old %ebp
Old %ebx

Set up

pushl %ebp
movl %esp,%ebp
pushl %ebx
Swap body

Stack entering swap

<table>
<thead>
<tr>
<th>Offset relative to %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Resulting stack

<table>
<thead>
<tr>
<th>Offset relative to %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Body

```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```
Swap finish

Stack at end swap body

...  
$zip2  
$zip1  
Return address  
Old %ebp  
Old %ebx

movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret  

Finish
Swap finish

Stack at end swap body

... $zip2 $zip1
Return address Old %ebp
%ebp → Old %ebx
%esp →

Resulting stack

... $zip2 $zip1
Return address Old %ebp
%ebp → Old %ebx
%esp →

Finish

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Swap finish

Stack at end swap body

```
... $zip2 $zip1
Return address
%ebp → Old %ebp
%esp → Old %ebx
```

Resulting stack

```
... $zip2 $zip1
Return address
%ebp → Old %ebp
%esp → Old %ebx
```

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Restores old value of %ebx!
Swap finish

Stack at end swap body

Resulting stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
Swap finish

Stack at end swap body

...  
$zip2  
$zip1  
Return address  

%ebp → Old %ebp  
%esp → Old %ebx

Resulting stack

%ebp →  
$zip2  
$zip1  
Return address  
%esp →

movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
Swap finish

Stack at end swap body

- ...  
- $zip2
- $zip1
- Return address
- Old %ebp
- Old %ebx

Resulting stack

- %ebp
- $zip2
- %esp
- $zip1

Finish

```
  movl -4(%ebp),%ebx
  movl %ebp,%esp
  popl %ebp
  ret
```
leave instruction

- Actual disassembly of swap

```
080483a4 <swap>:
80483a4:  55    push    %ebp
80483a5:  89 e5  mov      %esp,%ebp
80483a7:  53    push    %ebx
80483a8:  8b 55 08  mov 0x8(%ebp),%edx
80483ab:  8b 4d 0c  mov 0xc(%ebp),%ecx
80483ae:  8b 1a  mov (%edx),%ebx
80483b0:  8b 01  mov (%ecx),%eax
80483b2:  89 02  mov %eax,(%edx)
80483b4:  89 19  mov %ebx,(%ecx)
80483b6:  5b    pop %ebx
80483b7:  c9    leave
80483b8:  c3    ret
```

- `leave` prepares the stack for returning
- `leave` is equivalent to `movl %ebp,%esp
  popl %ebp
  ret`
Stack frame cheat sheet

```c
foo() {
    bar(38, 42);
}

bar(int arg1, int arg2) {
    int local1, local2;
    ...
    baz(55, 77);
}

baz(int arg1, int arg2) {
    ...
}
```
Return values

- By convention, the compiler leaves return value in %eax

```c
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8
```
Return values

- By convention, the compiler leaves return value in %eax

```
int logical(int x, int y)  
{  
    int t1 = x^y;  
    int t2 = t1 >> 17;  
    int mask = (1<<13) - 7;  
    int rval = t2 & mask;  
    return rval;  
}
```

- Works fine for 32-bit values
- For floating point values: other registers used
- For structs: return value is left on stack, caller must copy data elsewhere
  - Why must caller copy the data?
Register saving conventions

• When procedure `foo()` calls `bar()`
  `foo()` is the **caller**, `bar()` is the **callee**

• Suppose `bar()` needs to modify some registers when it runs
  • But `foo()` is using some of the same registers for its own purposes

```assembly
foo:
    ...
    movl $2138, %edx
    call bar
    addl %edx, %eax
    ...
    ret

bar:
    ...
    movl 8(%ebp), %edx
    addl $14850, %edx
    ...
    ret
```

• Contents of `%edx` clobbered by `bar()`!
Register saving conventions

- Need to save some of the clobbered registers on the stack.

- Who saves the registers? The caller? The callee?
  - **Caller save:** caller saves registers in its stack frame before call
  - **Callee save:** callee saves registers it will clobber in its stack frame, and restores them before return

- What are advantages and disadvantages of each?
  - Caller save: caller must be conservative and save everything, since it doesn’t know what callee will clobber.
  - Callee save: callee must be conservative and save everything, since it doesn’t know what caller wants preserved.
x86/Linux register conventions

- x86/Linux uses a mixture of caller-save and callee-save!
- Three registers managed as caller-save
  - %eax, %ecx, %edx
- Three registers managed as callee-save
  - %ebx, %esi, %edi
- Frame and stack registers managed specially
  - %esp, %ebp
Procedures summary

• The stack makes function calls work!
  • Private storage for each invocation of a procedure call
  • Multiple function invocations don't clobber each other
  • Addressing of local variables and arguments is relative to stack frame %ebp
  • Recursion works too
  • Requires that procedures return in order of invocations (nesting is preserved)

• Procedures implemented using a combination of hardware support plus software conventions
  • Hardware support: call, ret, leave, pushl, popl
  • Software conventions: Register saving conventions, managing %esp and %ebp, managing layout of stack
    • Software conventions defined by the OS and the compiler.
    • No guarantee it will be the same on a different software platform.
Today

- Procedures
  - The stack
  - Stack frames
  - Leave
  - Register conventions
- x86_64
x86-64

- x86 (aka IA32) instruction set defined in about 1985
  - Has been dominant instruction format for many years
- x86-64 extends x86 to 64 bits
  - Originally developed by AMD (Advanced Micro Devices), Intel’s competitor
    - Intel originally introduced Itanium (aka IA-64), a 64-bit ISA that was not backwards compatible. Not commercially successful.
  - Also referred to as AMD64, Intel64, and x64
- Currently in transition from 32 bits to 64 bits
  - Most new machines you buy will be 64 bits
## Differences between x86 and x86-64

### Data types

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly code suffix</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>Word</td>
<td>w</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>Quad word</td>
<td>q</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>Quad word</td>
<td>q</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>Extended precision</td>
<td>t</td>
<td>10/12</td>
<td>10/16</td>
</tr>
</tbody>
</table>
### Differences between x86 and x86-64

**Registers**

- **x86** has 8 registers
- **x86-64** has 16 registers
  - Each is 64 bits
  - Extend existing registers and add new ones
  - Make `%ebp`/`%rbp` general purpose

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>
x86-64 instructions

• Long word \( l \) (4 Bytes) \( \leftrightarrow \) Quad word \( q \) (8 Bytes)

• New instructions:
  • \texttt{movl} \( \rightarrow \) \texttt{movq} \quad \texttt{addl} \( \rightarrow \) \texttt{addq} \quad \texttt{sall} \( \rightarrow \) \texttt{salq} etc.

• 32-bit instructions generate 32-bit results
  • Set higher order bits of destination register to 0
  • E.g., \texttt{addl}

• \texttt{gcc} makes more efficient use of x86-64 instructions
  • E.g., more extensive use of conditional move operation

• \texttt{gcc} \ -m32 will produce 32-bit code
Procedure calls

• Up to six (integral) arguments can be passed in registers
  • Instead of on stack
  • %rdi, %rsi, %rdx, %rcx, %r8, %r9

• Some procedures do not need a stack frame at all!
  • Few arguments,
    few local variables,
    no local arrays or structs,
    no need to take address of local variables,
    no need to pass arguments on stack to another function,
    ⇒ no need for stack frame

• Can result in very low overhead for some function calls!
Stack frames

• No frame pointer!
  • x86_64 makes %rbp/%ebp general purpose

• Instead, procedures subtract a constant from stack pointer (%rsp) at beginning, add constant at procedure return
  • Accesses all stack elements via offsets from %rsp
  • No need for %rbp

• Stack frame size is constant during procedure call
  • Stack pointer does not fluctuate as in IA32
    • i.e., through pushes and pops
Next week

• Structures and arrays
• Buffer overruns
• Assignment 2 due Thursday