Credits and Disclaimers

The examples and discussion in the following slides have been adapted from a variety of sources, including:

Chapter 3 of Computer Systems 3rd Edition by Bryant and O'Hallaron
x86 Assembly/GAS Syntax on WikiBooks
(http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax)
Using Assembly Language in Linux by Phillip ??
(http://asm.sourceforge.net/articles/linasm.html)

The C code was compiled to assembly with gcc version 4.8.3 on CentOS 7.
Unless noted otherwise, the assembly code was generated using the following command line:

```
gcc -S -m64 -fno-asynchronous-unwind-tables -O0 file.c
```

AT&T assembly syntax is used, rather than Intel syntax, since that is what the gcc tools use.
Shift Instructions

Shifting the representation of an integer

sall rightop, leftop

leftop = leftop << rightop  // C syntax!

sarl rightop, leftop

leftop = leftop >> rightop (preserves sign)

shll rightop, leftop

leftop = leftop << rightop (same as sall)

shrl rightop, leftop

leftop = leftop >> rightop (hi bits set to 0)
Left Shifts and Multiplication

Shifting an integer operand to the left by \( k \) bits is equivalent to multiplying the operand's value by \( 2^k \):

\[
sall 1, \%eax \quad \# \ eax = 2^{\ast}eax
\]

\[
sall 3, \%edx \quad \# \ edx = 8^{\ast}edx
\]

For example:

\[
\begin{array}{c}
\text{edx} \\
00000000 00000000 00000000 00000101
\end{array}
\]

\[
\begin{array}{c}
\text{edx} \\
00000000 00000000 00000000 00101000
\end{array}
\]

Since general multiplication is much more expensive (in time) than shifting bits, we should prefer using a shift-left instruction when multiplying by a power of 2.
Shifting an integer operand to the right by \( k \) bits might be expected to divide the operand's value by \( 2^k \):

\[
\text{shrl 1, } \%\text{eax} \quad \# \quad \text{eax} = \text{eax} / 2
\]

Recall that \text{shrl} shifts in 0's on the left; so this will indeed perform integer division by 2, provided the value in \text{eax} is interpreted as an \textit{unsigned} integer.

For example, if we have an 8-bit unsigned representation of \( 255_{10} \), the instruction above would perform the following transformation:

\[
1111 \ 1111 \quad \rightarrow \quad 0111 \ 1111
\]

So it would yield \( 127_{10} \), which is correct for integer division.
Logical Instructions

There are the usual logical operations, applied bitwise:

```assembly
andl rightop, leftop
  leftop = leftop & rightop  // C syntax!

orl rightop, leftop
  leftop = leftop | rightop

xorl rightop, leftop
  leftop = leftop ^ rightop

notl op
  op = ~op
```
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    // Code...
}
```

Calling a function causes the creation of a stack frame dedicated to that function.

The frame pointer register, rbp, points to the beginning of the stack frame for the currently-running function.

The stack pointer register, rsp, points to the last thing that was pushed onto the stack.

(As an optimization, %rsp may or may not actually be updated. More on this later).
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z * 48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

The first 6 function arguments are passed in registers, additional arguments are passed on the stack.

The arguments stored in registers are often moved somewhere else on the stack before any computations.

In this example:
- x is passed in register `%edi` and is moved to `-20(%rbp).
- y is passed in register `%esi` and is moved to `-24(%rbp).
- z is passed in register `%edx` and is moved to `-28(%rbp).
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

Mapping:

<table>
<thead>
<tr>
<th>address</th>
<th>x</th>
<th>y</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbp - 24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbp - 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x86 Assembly

```asm
movl   -24(%rbp), %eax     # eax = y
movl   -20(%rbp), %edx     # edx = x
addl   %edx, %eax          # eax = x + y
movl   %eax, -4(%rbp)      # t1 = x + y
```
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z * 48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

Mapping:
- Address: address
  - z: rbp - 28
  - t2: rbp - 8

Assembly code:
- `movl -28(%rbp), %edx` # edx = z
- `movl %edx, %eax` # eax = z
- `addl %eax, %eax` # eax = z + z = 2z
- `addl %edx, %eax` # eax = 2z + z = 3z
- `sall $4, %eax` # eax = (3z) << 4 = 3z * 16 = 48z
- `movl %eax, -8(%rbp)` # t2 = 48z
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z * 48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

Mapping:

<table>
<thead>
<tr>
<th></th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>rbp - 4</td>
</tr>
<tr>
<td>t3</td>
<td>rbp - 12</td>
</tr>
</tbody>
</table>

Assembly code:

- `movl -4(%rbp), %eax` # `eax = t1`
- `movzwl $ax, %eax` # `eax = t1 & 0xFFFF`
- `movl %eax, -12(%rbp)` # `t3 = t1 & 0xFFFF`
Aside: movzwl

You may have noticed the `movzwl` instruction:

```
. . .
movzwl $ax, %eax       # eax = t1 & 0xFFFF
. . .
```

This moves a zero extended (Z) word (16 bits) stored in `%ax` to `%eax`.

And is equivalent to `t1 & 0xFFFF` since that will zero out the high 16 bits in `%eax` preserving the rest.

We'll see other versions of this instruction later. There are different sizes (`movzb`) and there's are signed variants (`movsb`).

In this case, `movzwl` apparently offered a performance (or some other) advantage.
Arithmetic/Logic Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
```

Mapping:

<table>
<thead>
<tr>
<th></th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2</td>
<td>rbp - 8</td>
</tr>
<tr>
<td>t3</td>
<td>rbp - 12</td>
</tr>
<tr>
<td>t4</td>
<td>rbp - 16</td>
</tr>
</tbody>
</table>

Assembly:

```assembly
movl  -8(%rbp), %eax      # eax = t2
imull -12(%rbp), %eax     # eax = t2 * t3
movl  %eax, -16(%rbp)      # t4 = t2 * t3
```
Assembled Code

```assembly
 .file "arith.c"
 .text
 .globl arith
 .type arith, @function
 arith:
   pushq %rbp     # save old frame pointer
   movq %rsp, %rbp # move frame pointer to top
   movl %edi, -20(%rbp) # move arguments x, y, and z
   movl %esi, -24(%rbp)
   movl %edx, -28(%rbp)
   ...
   movl -16(%rbp), %eax # set return value in eax
   popq %rbp      # rsp = rbp; pop to rbp
   ret           # return to caller
 .size arith, -.arith
 .ident "GCC: (GNU) 4.8.3 20140911 ..."
```

```
gcc -O0 -S -Wall -m32 arith.c

Assembled Code

```

```
int arith(int x, int y, int z) {

  ...
  int t4 = t2 * t3;
  return t4;
}
```
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z * 48;
    ...
Assembled Code

```
int arith(int x, int y, int z) {

    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    ...}
```

```
movl -4(%rbp), %eax  # eax = t1
movzw %ax, %eax      # eax = t1 & 0xFFFF
movl %eax, -12(%rbp) # t3 = t1 & 0xFFFF

movl -8(%rbp), %eax  # eax = t2
imull -12(%rbp), %eax # eax = t2 * t3
movl %eax, -16(%rbp) # t4 = t2 * t3
```