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 -mno-red-zone -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

\[
\begin{align*}
\text{sall } & \text{ rightop, leftop} \\
\text{leftop} & = \text{leftop} \ll \text{rightop} \quad \text{// C syntax!} \nonumber \\
\text{sarl } & \text{ rightop, leftop} \\
\text{leftop} & = \text{leftop} \gg \text{rightop} \quad \text{(preserves sign)} \\
\text{shll } & \text{ rightop, leftop} \\
\text{leftop} & = \text{leftop} \ll \text{rightop} \quad \text{(same as sall)} \\
\text{shrl } & \text{ rightop, leftop} \\
\text{leftop} & = \text{leftop} \gg \text{rightop} \quad \text{(hi bits set to 0)}
\end{align*}
\]
Left Shifts and Multiplication

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

\[
\begin{align*}
\text{sall 1, } \%eax & \quad \# \text{ eax} = 2^1 \times \text{eax} \\
\text{sall 3, } \%edx & \quad \# \text{ edx} = 2^3 \times \text{edx}
\end{align*}
\]

For example:

<table>
<thead>
<tr>
<th>edx</th>
<th>00000000 00000000 00000000 00000101</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>00000000 00000000 00000000 00101000</td>
<td>40</td>
</tr>
</tbody>
</table>

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 $2^k$:

```
shrl 1, %eax     # eax = eax / 2 ?
```

Recall that `shrl` shifts in 0's on the left; so this will indeed perform integer division by 2, provided the value in `eax` is interpreted as an 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       \rightarrow       0111 1111
```

So it would yield $127_{10}$, which is correct for integer division.
But, the following will *not* yield the correct result for an unsigned integer:

\[
\text{sar}l \ 1, \ %\text{eax} \quad \# \ \text{eax} \neq \text{eax} / 2
\]

For example, if we consider an 8-bit representation of \(200_{10}\), the instruction above would produce this transformation:

\[
\begin{array}{c}
1100\ 1000 \\
\Rightarrow \\
1110\ 0100
\end{array}
\]

So it would yield \(228_{10}\), which is incorrect.

The correct result would be \(100_{10}\) which would be represented as \(0110\ 0010\).

Note that the correct value would have been found by using \text{shrl} instead.
Shifting a non-negative (signed) integer operand to the right by \( k \) bits will divide the operand's value by \( 2^k \):

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

If \( eax \) holds a non-negative signed integer, the left-most bit will be 0, and so both of these instructions will yield the same result.

But, if the signed operand is negative, then the high bit will be 1.

Clearly, \text{shrl} cannot yield the correct quotient in this case. Why?
Right Shifts, Signed Operands, and Division

What about the following instruction, if eax holds a negative signed value?

```
sarl 1, %eax # eax = eax / 2
```

sarl replicates the sign bit, so this will yield a negative result...

But, suppose we have an 8-bit representation of -7: 1111 1001

Then applying an arithmetic right shift of 1 position yields: 1111 1100

That represents the value -4… is that correct?

Mathematics says yes by the Division Algorithm:

-7 = -4 * 2 + 1

Remainders must be >= 0!

C says no:

-7 = -3 * 2 + -1

-7 % 2 must equal -(7 % 2)
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
```
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>rbp - 24</td>
<td>rbp - 4</td>
<td></td>
</tr>
</tbody>
</table>

Assembly Code:

- `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:

<table>
<thead>
<tr>
<th>address</th>
<th>mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>rbp - 28</td>
</tr>
<tr>
<td>t2</td>
<td>rbp - 8</td>
</tr>
</tbody>
</table>

Assembly code:

```assembly
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
```
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>

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.
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></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>

movl -8(%rbp), %eax  # eax = t2
imull -12(%rbp), %eax # eax = t2 * t3
movl %eax, -16(%rbp) # t4 = t2 * t3
.file "arith.c"
.text
.globl arith
.type arith, @function
arith:
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -20(%rbp)
    movl %esi, -24(%rbp)
    movl %edx, -28(%rbp)
    ...
    movl -16(%rbp), %eax
    popq %rbp
    ret
.size arith, .-arith
.ident "GCC: (GNU) 4.8.3 20140911 ...

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;
    . . .
}
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;
}

...