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 2nd 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 4.5.2 on Ubuntu Linux.

Unless noted otherwise, the assembly code was generated using the following command line:

```
 gcc -S -m32 -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

\[
\text{sall } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} \ll \text{rightop} \quad \text{// C syntax!}
\]

\[
\text{sarl } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} \gg \text{rightop} \quad \text{(preserves sign)}
\]

\[
\text{shll } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} \ll \text{rightop} \quad \text{(same as sall)}
\]

\[
\text{shrl } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} \gg \text{rightop} \quad \text{(hi bits set to 0)}
\]
Left Shifts and Multiplication

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

\[
\text{sal} \ 1, \ %eax \quad \# \ eax = 2^{*}eax
\]

\[
\text{sal} \ 3, \ %edx \quad \# \ edx = 8^{*}edx
\]

For example:

- eax
  - Original: 00000000 00000000 00000000 00000101
  - After shifting: 00000000 00000000 00000000 00101000

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.
Right Shifts, Unsigned Operands, and Division

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:

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

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

\[
1100\ 1000 \quad \rightarrow \quad 1110\ 0100
\]

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 `shrl` instead.
Shifting a non-negative (signed) integer operand to the right by k bits will divide the operand's value by $2^k$:

```
shrl 1, %eax  # eax = eax / 2
sar 1, %eax   # 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, `shrl` cannot yield the correct quotient in this case. Why?
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:

\[-7 = -4 \times 2 + 1\]

Remainders must be >= 0!

C says no:

\[-7 = -3 \times 2 + -1\]

\[-7 \% 2 \text{ must equal } -(7 \% 2)\]
Logical Instructions

There are the usual logical operations, applied bitwise:

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

orl rightop, leftop
   leftop = leftop | righttop

xorl rightop, leftop
   leftop = leftop ^ rightop

notl op
   op = ~op
```
int arith(int x, int y, int z) {
  ...
}

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

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

The *stack pointer register*, esp, points to the last thing that was pushed onto the stack.
Arithmetic/Logic Example

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 Stack

| ebp + 16 | . . . |
| ebp + 12 |
| ebp + 8  |
| ebp + 4  |
| ebp      |
| ebp - 4  |
| ebp - 8  |
| ebp - 12 |
| ebp - 16 |

params passed to fn
autos within fn

return address
old value of ebp
t1
t2
t3
t4

z
y
x

CS@VT
Computer Organization I

©2005-2012 McQuain
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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>ebp + 8</td>
</tr>
<tr>
<td>y</td>
<td>ebp + 12</td>
</tr>
<tr>
<td>t1</td>
<td>ebp - 4</td>
</tr>
</tbody>
</table>

Assembly Code:

```assembly
movl 12(%ebp), %eax          # eax = y
movl 8(%ebp), %edx           # edx = x
lea  (%edx,%eax), %eax       # eax = x + y
movl %eax, -4(%ebp)          # t1 = x + y
```
Aside: leal

You also noticed another use of the `leal` instruction:

```assembly
leal (%eax,%eax), %edx  # edx = eax + eax
```

The particular form of the instruction used here is:

```
leal (src1, src2), dst  
dst = src2 + src1
```

We'll see other versions of this instruction later.

The execution of the instruction offers some additional performance advantages.
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>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp + 16</td>
<td>z</td>
</tr>
<tr>
<td>ebp - 8</td>
<td>t2</td>
</tr>
</tbody>
</table>

assembly code:

```
movl 16(%ebp), %edx  # edx = z
movl %edx, %eax      # eax = z
addl %eax, %eax      # eax = 2z
addl %edx, %eax      # eax = 3z
sall $4, %eax        # eax = (3z) << 4 = 48z
movl %eax, -8(%ebp)  # t2 = 48z
```
Aside: Optimization

You noticed something interesting about the way the compiler translated the C statement:

```
int t2 = z * 48;
```

The resulting assembly code did not use `imull`:

```
movl 16(%ebp), %eax          # eax = z
leal (%eax,%eax,2), %edx     # edx = eax + 2*eax
sall $4, %edx                # edx = 16*edx
```

The original C code is equivalent to:

```
int t2 = (z + z << 1) << 4;
```

Recall that shifting is equivalent to multiplying by a power of 2.

It is also faster to shift (and add) than to multiply, so the translation above saves some time.
Aside: leal

You also noticed another use of the `leal` instruction:

```assembly
leal (%eax,%eax,2), %edx       # edx = eax + 2*eax
```

The particular form of the instruction used here is:

```
leal Imm1(src1, src2, Imm2), dst
```
```
dst = Imm2*src2 + src1 + Imm1
```

The execution of the instruction offers some additional performance advantages.
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>EBP - 4</th>
<th>EBP - 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly:

```assembly
movl -4(%ebp), %eax       # eax = t1
andl $65535, %eax         # eax = t1 & 0xFFFF
movl %eax, -12(%ebp)      # t3 = t1 & 0xFFFF
```
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>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp - 8</td>
<td>ebp - 12</td>
<td>ebp - 16</td>
<td></td>
</tr>
</tbody>
</table>

Assembly Code:

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

```assembly
.int arith(int x, int y, int z) {
  . . .
  int t4 = t2 * t3;
  return t4;
}
gcc -O0 -S -Wall -m32 arith.c
```
Assembled Code

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z*48;
    . . .
}
```

```x86 Assembly
...    movl 12(%ebp), %eax       # eax = y
    movl 8(%ebp), %edx        # edx = x
    leal (%edx,%eax), %eax    # eax = x + y
    movl %eax, -4(%ebp)       # t1 = x + y

    movl 16(%ebp), %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(%ebp)       # t2 = 48z
...```

CS@VT
Computer Organization I
©2005-2012 McQuain
Assembled Code

```assembly
movl  -4(%ebp), %eax    # eax = t1
andl $65535, %eax      # eax = t1 & 0xFFFF
movl %eax, -12(%ebp)   # t3 = t1 & 0xFFFF

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

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

    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    ...
}
```