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.
Program Translation Overview

- **text**
  - C program (p1.c p2.c)
    - Compiler (gcc -S)
  - Asm program (p1.s p2.s)
    - Assembler (gcc or as)

- **binary**
  - Object program (p1.o p2.o)
    - Linker (gcc or ld)
  - Executable program (p)
    - Static libraries (.a)
X86-32 Integer Registers

- %eax, %ax, %ah, %al
- %ecx, %cx, %ch, %cl
- %edx, %dx, %dh, %dl
- %ebx, %bx, %bh, %bl
- %esi
- %edi
- %esp, %sp
- %ebp, %bp

Origin (mostly obsolete):
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer

16-bit virtual registers
### x86-64 Integer Registers

<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>

- Extend existing registers. Add 8 new ones.
- Make %ebp/%r bp general purpose
Due to the long history of the x86 architecture, the terminology for data lengths can be somewhat confusing:

- **byte** \( b \) 8 bits, no surprises there
- **short** \( s \) 16-bit integer or 32-bit float
- **word** \( w \) 16-bit value
- **long** \( l \) 32-bit integer or 64-bit float (aka double word)
- **quad** \( q \) 64-bit integer

The single-character abbreviations are used in the names of many of the x86 assembly instructions to indicate the length of the operands.

As long as the widths of the operands match, any of these suffixes can be used with the assembly instructions that are discussed in the following slides; for simplicity, we will generally restrict the examples to operations on **long** values.
Simple Example: C to Assembly

```assembly
.globl main
.type main, @function

main:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $5, -4(%rbp)
movl $16, -8(%rbp)
movl -8(%rbp), %eax
movl -4(%rbp), %edx
addl %edx, %eax
movl %eax, -12(%rbp)
movl $0, %eax
popq %rbp
ret

.size main, .-main
.ident "GCC: (GNU) 4.8.3 20140911 (Red Hat 4.8.3-9)"
.section .note.GNU-stack,"",@progbits
```

```
int main() {
    int x, y, t;
    
    x = 5;
y = 16;
t = x + y;
    
    return 0;
}
```

gcc -O0 -S -Wall -m64 simplest.c
Simple Example: Memory Layout

Local variables and function parameters are stored in memory, and organized in a stack frame.

Two registers are used to keep track of the organization:

- `rsp` address of the top element on the stack
- `rbp` address of the first element in the current stack frame

```c
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}
```

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td>old value of rbp</td>
</tr>
<tr>
<td>rbp - 4</td>
<td>x</td>
</tr>
<tr>
<td>rbp - 8</td>
<td>y</td>
</tr>
<tr>
<td>rbp - 12</td>
<td>t</td>
</tr>
<tr>
<td>rsp</td>
<td></td>
</tr>
</tbody>
</table>
Register-Memory Data Transfers

Many machine-level operations require that data be transferred between memory and registers.

The most basic instructions for this are the variants of the `mov` instruction:

```assembly
movl src, dest
dest := src
```

This copies a 32-bit value from `src` into `dest`. `movq` moves 64 bit values in the same fashion.

Despite the name, it has no effect on the value of `src`.

The two operands can be specified in a number of ways:

- immediate values
- one of the 16 x86-64 integer registers (or their virtual registers)
- memory address
Operand Specifications

**Immediate:** Constant integer data
Example: \$0x400, \$-533
Like C constant, but prefixed with ‘\$’
Encoded with 1, 2, or 4 bytes

**Register:** One of the 16 integer registers
Example: %eax, %edx (reg names preceded by ’%’)  
But %rsp and %rbp reserved for special use  
Others have special uses for particular instructions

**Memory:** \(N\) consecutive bytes of memory at address given by register, \(N\) is specified by the instruction name, \texttt{movl} = 4 bytes, \texttt{movq} = 8 bytes.  
Simplest example: (%rax)  
Various other “address modes”
## Basic Examples

<table>
<thead>
<tr>
<th>X86-64 Assembly</th>
<th>C analog</th>
<th>Mapping:</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0x10, %eax</td>
<td>a = 16;</td>
<td>reg</td>
</tr>
<tr>
<td>movl $42, %ebx</td>
<td>b = 42;</td>
<td>a %eax</td>
</tr>
<tr>
<td>movl %ecx, %edx</td>
<td>d = c;</td>
<td>b %ebx</td>
</tr>
<tr>
<td>movl %eax, (%rbx)</td>
<td>*b = a</td>
<td>c %ecx</td>
</tr>
<tr>
<td>movl (%rbx), %eax</td>
<td>a = *b</td>
<td>d %edx</td>
</tr>
<tr>
<td>movl -4(%rbp), %eax</td>
<td>a = *(rbp - 4)</td>
<td></td>
</tr>
</tbody>
</table>
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}

```assembly
movl   $5, -4(%rbp)
movl   $16, -8(%rbp)
movl   -8(%rbp), %eax
addl   %edx, %eax
movl   %eax, -12(%rbp)
```
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}

movl $5, -4(%rbp)

<table>
<thead>
<tr>
<th>rbp</th>
<th>old value of rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 4</td>
<td>5</td>
</tr>
<tr>
<td>rbp - 8</td>
<td>??</td>
</tr>
<tr>
<td>rbp - 12</td>
<td>??</td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>eax</th>
<th>??</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>??</td>
</tr>
</tbody>
</table>
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}

define

movl $5, -4(%rbp)
movl $16, -8(%rbp)

<table>
<thead>
<tr>
<th>rbp</th>
<th>old value of rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 4</td>
<td>5</td>
</tr>
<tr>
<td>rbp - 8</td>
<td>16</td>
</tr>
<tr>
<td>rbp - 12</td>
<td>??</td>
</tr>
</tbody>
</table>

Registers

<table>
<thead>
<tr>
<th>eax</th>
<th>??</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>??</td>
</tr>
</tbody>
</table>

the Stack
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}

movl $5, -4(\%rbp)
movl $16, -8(\%rbp)
movl -8(\%rbp), \%eax
movl -4(\%rbp), \%edx
addl \%edx, \%eax
movl \%eax, -12(\%rbp)
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}

```
    movl $5, -4(%%rbp)
    movl $16, -8(%%rbp)
    movl -8(%%rbp), %eax
    movl -4(%%rbp), %edx
    addl %edx, %eax
    movl %eax, -12(%%rbp)
```
```c
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}
```

```
movl $5, -4(%rbp)
movl $16, -8(%rbp)
movl -8(%rbp), %eax
movl -4(%rbp), %edx
addl %edx, %eax
movl %eax, -12(%rbp)
```
Integer Arithmetic Instructions

We have the expected addition operation:

\[
\text{addl } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} + \text{rightop}
\]

The operand ordering shown here is probably confusing:

- As usual, the destination is listed second.
- But, that's also the first (left-hand) operand when the arithmetic is performed.

This same pattern is followed for all the binary integer arithmetic instructions.

See the discussion of AT&T vs Intel syntax later in the notes for an historical perspective on this.
```c
int main() {
    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}
```

### Registers

<table>
<thead>
<tr>
<th>eax</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>5</td>
</tr>
</tbody>
</table>

### Stack

- **rbp**
  - **old value of rbp**: 5
  - **rbp - 4**: 5
  - **rbp - 8**: 16
  - **rbp - 12**: 21

### X86-64 Assembly

- `movl $5, -4(%rbp)`
- `movl $16, -8(%rbp)`
- `movl -8(%rbp), %eax`
- `movl -4(%rbp), %edx`
- `addl %edx, %eax`
- `movl %eax, -12(%rbp)`
More Arithmetic Instructions

In addition:

\[
\text{subl } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} - \text{rightop}
\]

\[
\text{imull } \text{rightop, leftop} \\
\text{leftop} = \text{leftop} \times \text{rightop}
\]

\[
\text{negl } \text{op} \\
\text{op} = -\text{op}
\]

\[
\text{incl } \text{op} \\
\text{op} = \text{op} + 1
\]

\[
\text{decl } \text{op} \\
\text{op} = \text{op} - 1
\]

(Yes, there is a division instruction, but its interface is confusing and we will not need it.)