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)

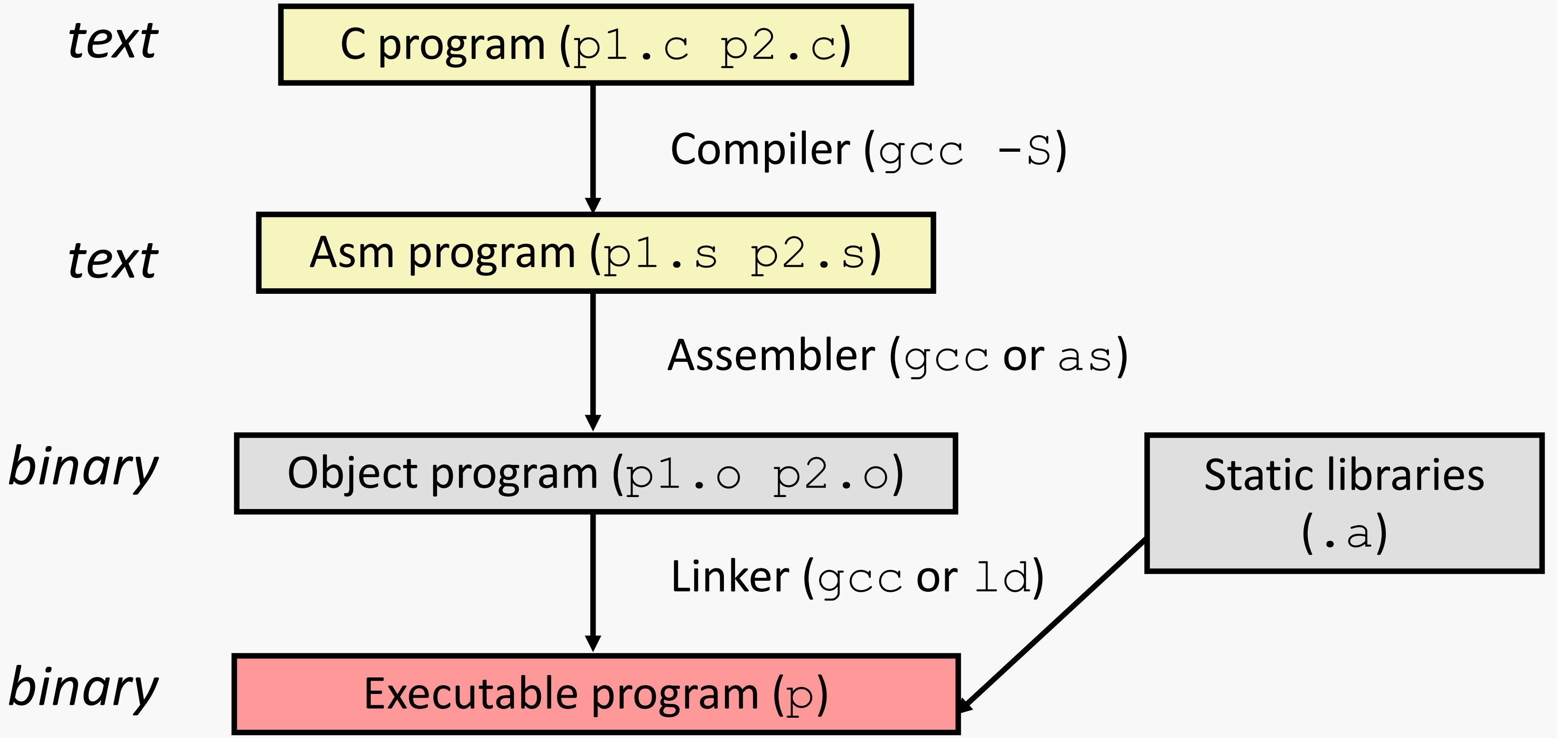
The C code was compiled to assembly with `gcc` version 4.5.2 on Ubuntu Linux.

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

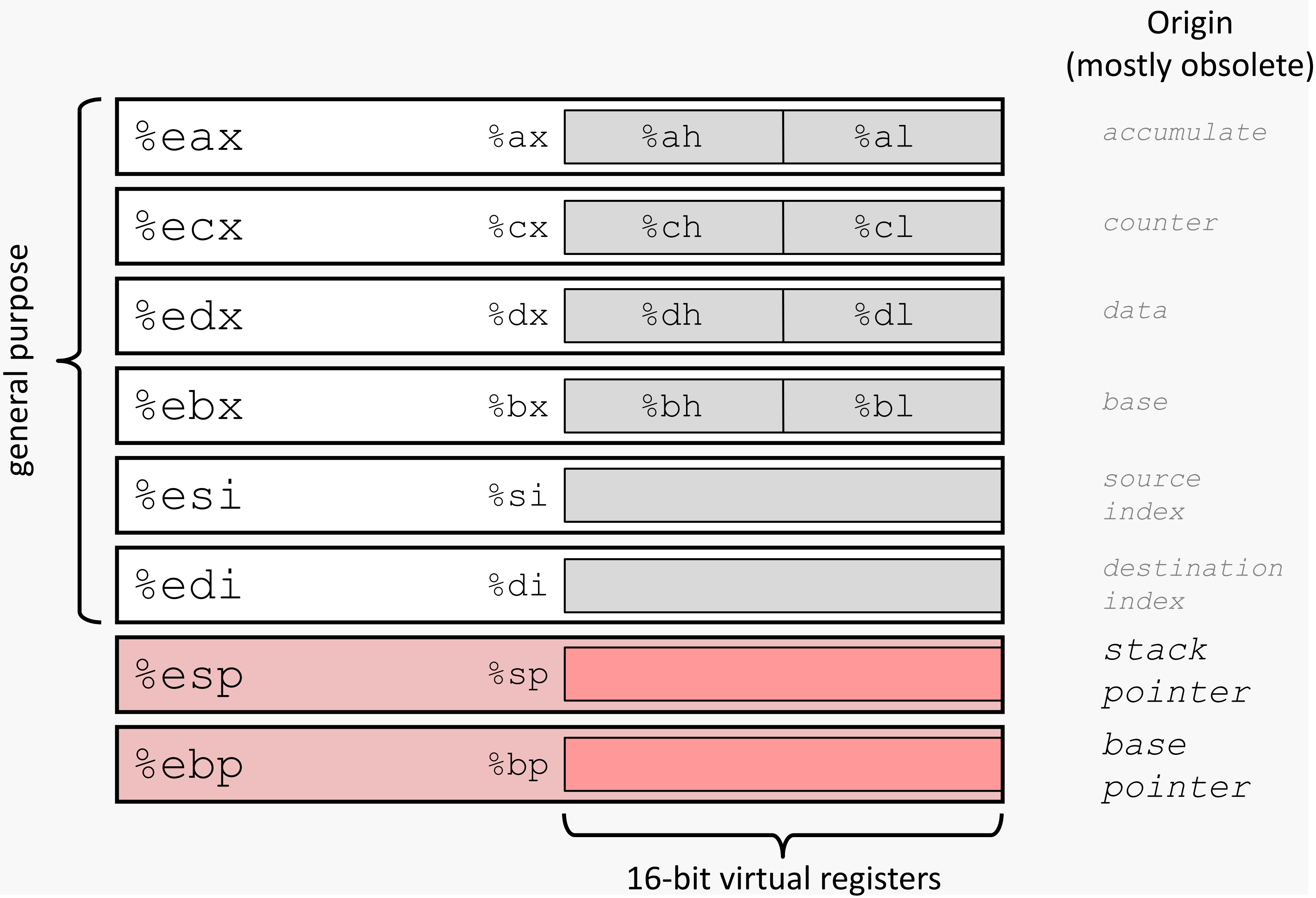
```
gcc -S -m32 -O0 file.c
```

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

Program Translation Overview



IA-32 Integer Registers



IA-32 Nomenclature

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

```
gcc -O1 -S -Wall -m32 simplest.c
```

```
        .file   "simplest.c"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $5, -4(%ebp)
        movl    $16, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    -4(%ebp), %edx
        leal    (%edx,%eax), %eax
        movl    %eax, -12(%ebp)
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident "GCC: (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2"
        .section        .note.GNU-stack,"",@progbits
```

```
int main() {

   int x, y, t;

   x = 5;
   y = 16;
   t = x + y;

   return 0;
}
```

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:

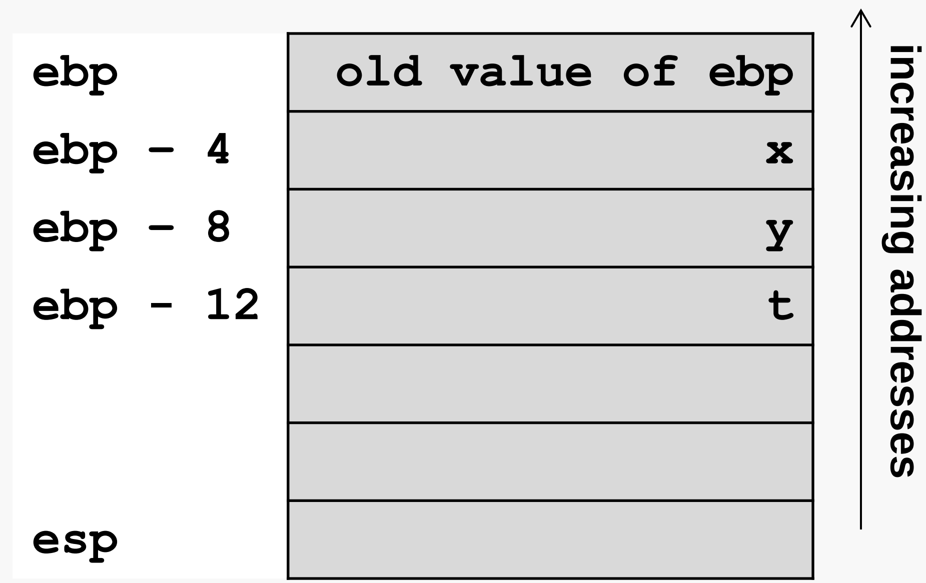
`esp`	address of the top element on the stack
`ebp`	address of the first element in the current stack frame

```
int main() {

   int x, y, t;

   x = 5;
   y = 16;
   t = x + y;

   return 0;
}
```



the Stack

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:

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

This copies a 32-bit value from *`src`* into *`dest`*.

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 8 IA-32 integer 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 8 integer registers

Example: **`%eax, %edx (reg names preceded by '%')`**

But **`%esp`** and **`%ebp`** reserved for special use

Others have special uses for particular instructions

Memory: 4 consecutive bytes of memory at address given by register

Simplest example: **`(%eax)`**

Various other "address modes"

Basic Examples

x86 assembly	C analog
`movl $0x10, %eax`	`a = 16;`
`movl $42, %ebx`	`b = 42;`
`movl %ecx, %edx`	`d = c;`
`movl %eax, (%ebx)`	`*b = a`
`movl (%ebx), %eax`	`a = *b`

```
Mapping:
        reg
a       %eax
b       %ebx
c       %ecx
d       %edx
```

C to Assembly

```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	old value of ebp
ebp - 4	x
ebp - 8	y
ebp - 12	t

the Stack

```
movl   $5, -4(%ebp)
```

```
movl   $16, -8(%ebp)
```

```
movl   -8(%ebp), %eax
movl   -4(%ebp), %edx
leal   (%edx,%eax), %eax
movl   %eax, -12(%ebp)
```

Registers

eax	
ebx	
ecx	
edx	
edi	
esi	

C to Assembly

```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	**old value of ebp**
ebp - 4	**5**
ebp - 8	**??**
ebp - 12	**??**

the Stack

```
movl   $5, -4(%ebp)
```

Registers

eax	**??**
edx	**??**

C to Assembly

```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	old value of ebp
ebp - 4	5
ebp - 8	16
ebp - 12	??

the Stack

```
movl   $5, -4(%ebp)
```

```
movl   $16, -8(%ebp)
```

Registers

eax	??
edx	??

C to Assembly


```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	**old value of ebp**
ebp - 4	**5**
ebp - 8	**16**
ebp - 12	**??**

the Stack

```
movl   $5, -4(%ebp)
```

```
movl   $16, -8(%ebp)
```

```
movl   -8(%ebp), %eax
movl   -4(%ebp), %edx
leal   (%edx,%eax), %eax
movl   %eax, -12(%ebp)
```

Registers

eax	**16**
edx	**??**

C to Assembly


```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	old value of ebp
ebp - 4	5
ebp - 8	16
ebp - 12	??

the Stack

```
movl   $5, -4(%ebp)
```

```
movl   $16, -8(%ebp)
```

```
movl   -8(%ebp), %eax
movl   -4(%ebp), %edx
leal   (%edx,%eax), %eax
movl   %eax, -12(%ebp)
```

Registers

eax	16
edx	5

C to Assembly

```
int main() {

    int x, y, t;
    x = 5;
    y = 16;
    t = x + y;
    return 0;
}
```

ebp	old value of ebp
ebp - 4	5
ebp - 8	16
ebp - 12	??

the Stack

```
movl    $5, -4(%ebp)
```

```
movl    $16, -8(%ebp)
```

```
movl    -8(%ebp), %eax
movl    -4(%ebp), %edx
leal    (%edx,%eax), %eax
movl    %eax, -12(%ebp)
```

Registers

eax	21
edx	5

Aside: leal

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

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

The particular form of the instruction used here on the previous slide is:

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

The execution of the instruction offers some additional performance advantages.

C to Assembly

```
int main() {

   int x, y, t;
   x = 5;
   y = 16;
   t = x + y;
   return 0;
}
```

ebp	old value of ebp
ebp - 4	5
ebp - 8	16
ebp - 12	21

the Stack

```
movl   $5, -4(%ebp)
```

```
movl   $16, -8(%ebp)
```

```
movl   -8(%ebp), %eax
movl   -4(%ebp), %edx
leal   (%edx,%eax), %eax
movl   %eax, -12(%ebp)
```

Registers

eax	21
edx	5

Integer Arithmetic Instructions

We have the expected addition operation:

```
addl rightop, leftop
      leftop = leftop + 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.

More Arithmetic Instructions

In addition:

```
subl rightop, leftop
      leftop = leftop - rightop

imull rightop, leftop
      leftop = leftop * rightop

negl op
      op = -op

incl op
      op = op + 1

decl op
      op = op - 1
```

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